VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Editor pro GLSL

GLSL Editor

2014

Petr Buček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Buček**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Editor pro GLSL
GLSL Editor

Zásady pro vypracování:

Cílem práce je implementovat editor pro grafické shadery, primárně GLSL. Základní komponentou editoru bude jednoduchý správce, který bude umožňovat načítání modelů ve vybraném formátu, specifikovat různé elementy scény, jako např. světla, materiály, apod. Textová, editační část pro přímé psaní shaderů bude umožňovat zvýrazňování klíčových slov, komentářů, apod. Součástí aplikace bude demonstrační 3D scéna, ve které budou vidět aplikované změny po kompilaci shaderu. Doporučeným implementačním jazykem je C++.
Body zadání:
1. Shrnutí současného stavu a existujících aplikací pro editace grafických shaderů s bližším zaměřením na GLSL.
2. Návrh a implementace vlastního editoru
3. Výkonnostní testy aplikace a jejich zhodnocení.
4. Vytvoření ukázkových shaderů.

Seznam doporučené odborné literatury:

[1] John Kessenich: The OpenGL Shading Language, http://www.opengl.org/documentation/glsl/, 2011
[2] Mark Segal and Kurt Akeley: The OpenGL Graphics System: A Specification,
2011

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2014

Bučok Peter

Poděkování
Tato práce byla vypracována s podporou projektu Bio-inspirované metody: věda, vzdělávání a transfer znalostí, reg. č. CZ.1.07/2.3.00/20.0073 podpořeného Operačním programem Vzdělávání pro konkurenceschopnost, financovaného ze strukturálních fondů EU a státního rozpočtu ČR.

Rád bych poděkoval všem, kteří pomáhali se vznikem této práce, ať už svými znalostmi, nebo svou trpělivostí.

Abstrakt

Cílem práce je vytvořit moderní nástroj pro zjednodušení vývoje shaderů v jazyce OpenGL Shading Language, včetně porovnání a zhodnocení existujících řešení. Důraz byl kladen na jednoduchost použití vzniklého nástroje a možnost jeho využití při výuce shaderů. Nástroj podporuje doplňování kódu, vizualizaci vytvářeného shaderu i udržení ukázkové scény jako celku. Součástí práce je i sada demonstračních scén spolu s výsledky testů vzniklého programu.

Klíčová slova: C#, .NET, C99, OpenGL, GLSL, shader, editor

Abstract

The aim of the thesis is to create modern tool for simplification of shaders development in OpenGL Shading Language, including comparison and evaluation of existing solutions. The emphasis was on using simplicity of the tool and on possibility of using it for shaders lessons in education. The tool supports code complementarity, visualization of generated shader and maintaining of demonstration scene like a whole. Part of the thesis is also set of demonstration scenes, including test results of created program.

Keywords: C#, .NET, C99, OpenGL, GLSL, shader, editor

Seznam použitých zkratek a symbolů

AGL	–	Rozšíření pro Mac OS
ARB	–	Architecture Review Board
ASCII	–	American Standard Code for Information Interchange
AWT	–	Abstract Window Toolkit
Cg	–	C for Graphics
CLI	–	Command Line Interface
CPU	–	Centrální procesor
GLSL	–	OpenGL Shading Language
GLX	–	Rozšíření pro X-Server
GPU	–	Grafický procesor
HLSL	–	High Level Shading Language
HTML5	–	Hypertext Markup Language 5
IDE	–	Integrated Development Environment
IEEE 754	–	Standard pro dvojkovou aritmetiku v pohyblivé řádové čárce
JOGL	–	Java OpenGL
JSON	–	JavaScript Object Notation
LGPL	–	Lesser General Public Library
LWJGL	–	Light Weight Java Game Library
OpenGL	–	Open Graphics Library
NURBS	–	Non-Uniform Rational B-spline
TCP/IP	–	Rodina síťových protokolů
UTF-8	–	UNICODE Text Format
WGL	–	Rozšíření pro Windows
WPF	–	Windows Presentation Foundation
XML	–	eXtensible Markup Language
XAML	–	Extensible Application Markup Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Vykreslování obrazu z matematických dat je dnes velmi často požadovaná úloha, skrývající rozsáhlé výpočetní pozadí. Takovouto vizualizaci dat potřebují různé vědní obory od lékařství, přes architekturu, až po zábavní průmysl. Pro tento účel se časem vyvinul specializovaný hardware zvaný grafická karta, umožňující urychlení těchto výpočtů.

Začátek této práce je věnován shrnutí vlastností a vývoji normy OpenGL. Obsahuje základní popis věcí a úloh nutný k lepšímu pochopení celé práce.

Před zahájením vývoje vlastního editoru bylo potřeba zhodnotit aktuální možnosti práce se shadery. Toto téma je rozvedeno v kapitole 2.3, kde lze nalézt shrnutí informací o zkoumaných editorech.

Jelikož má editor sloužit k úpravě a vývoji shaderů jazyka GLSL, bylo důležité prozkoumat a zdokumentovat klíčové prvky jazyka. Kapitolu 3 lze považovat za stručný výčet takovýchto prvků. Důraz je pak kladen na popis částí, klíčových při vývoji shader editoru.

Po získání všech potřebných informací započal samotný vývoj editoru. První fází bylo vybrat technologie pro vývoj a navrhnout celkové fungování programu. Chování editoru bylo rozmyšleno i s ohledem na uživatelskou přívětivost a využití funkcí, které hostující operační systém systém nabízí.

Druhou fází je pak samotná implementace, kde je popsán technický způsob řešení problematických nebo zajímavých částí práce. Důraz byl kladen na modularitu a případnou rozšiřitelnost celého řešení.

Na závěr je vzniklý editor podroben několika testům a doplněn o krátké zhodnocení.

2 Základy vykreslování obrazu

Grafickou kartu, potažmo OpenGL, lze chápat jako symbolický stroj, do kterého vstupují matematická data a vystupuje jejich obraz. Samotný proces vizualizace je pak rozdělen do několika kroků, které se v průběhu let měnily a zobecňovaly.

Obrázek 1: Schéma činnosti grafické karty

Původní grafické karty využívaly odlišná rozhraní, jakými byly IrisGL nebo PHIGS [1]. Nevýhodou byla uzavřenost (rozhranní vyvinuté nebo podporované pouze jednou firmou) nebo zbytečná složitost takových řešení. Postupem času se navíc nároky na tento hardware stále zvyšovaly, a stejně tak bylo složitější přenášet vyvíjené aplikace mezi jednotlivými systémy a platformami. To vyústilo v potřebu jednotného přístupu k práci s obrazem, kterým se stala norma OpenGL.

Cílem bylo vytvořit normu pro výrobce hardwaru a sjednotit tak programový přístup k vykreslování obrazu s hardwarovou akcelerací. OpenGL vzniklo v roce 1992 jako otevřený standart z proprietárního rozhranní IrisGL firmy SGI, a bylo vyvíjeno a udržováno sdružením **OpenGL Architectural Review Board**. Na podzim roku 2006 bylo OpenGL předáno **Khronos Group**, která se o tento standard stará dodnes. [2]

Pod skupinu Khronos dnes spadají technologie jako OpenGL ES (varianta OpenGL pro mobilní zařízení), WebGL (varianta OpenGL pro technoligii HTML5) nebo souborový formát pro přenos 3D scén COLLADA, vyvinutý původně společností SONY pro PlayStation 3. Khronos pracuje i na prosazení technologie EGL, která je platformově nezávislá a nahrazuje rozhranní jako GLX (propojení OpenGL do technologie X-Server), WGL (obdoba pro Windows) nebo AGL (varianta společnosti Apple). [2]

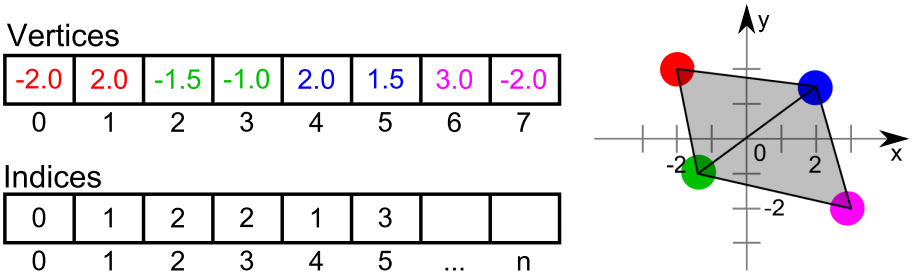
2.1 Programovatelný řetězec

Jak již bylo řečeno, cesta k vykreslení obrazu se skládá z jednotlivých kroků, které dohromady tvoří takzvaný vykreslovací řetězec. Tím prochází data, která se postupně transformují ve výsledný obraz.

Vstupem na začátku řetězce jsou programátorem zadané vrcholy, včetně jejich vlastností a informace o tom, jak tyto vrcholy tvoří primitiva. Primitivem se rozumí libovolný polygon (současné grafické karty jej rozloží na trojúhelníky), přímo trojúhelníky nebo z nich složená posloupnost (pruh nebo vějíř). Ve speciálních případech může jít i o jiné než rovinné útvary, jakými jsou čáry (úsečky) nebo body. Složitější struktury jako jsou NURBS, křivky apod. v rámci této práce nebudou uvažovány.

Vrcholem se v tomto případě rozumí prvek, který má pozici v trojrozměrném prostoru a sadu vlastností. Při zanedbání určitých souřadnic lze uvažovat i bod ve dvou nebo jednorozměrném prostoru. Vlastnosti byly v prvních verzích dány pevně hardwarem a jednalo se o:

- Primární barvu
- Sekundární barvu
- Normálový vektor
- Souřadnici textury
- Souřadnici mlhy
- Index barvy
- Značku hrany



Obrázek 2: Způsob skládání trojúhelníků

Během vývoje se tyto vlastnosti zobecnily a dnes je možné k vrcholu navázat libovolnou vlastnost s ohledem na použitý shader. Vlastnost je zde reprezentována pomocí čísla, vektoru nebo matice (ať už celočíselné nebo s plovoucí řádovou čárkou).

Data do řetězce nevstupují pouze na začátku, ale i v jeho průběhu. Takovým příkladem dat jsou například textury.

Stejně jako vlastnosti vrcholů se i samotný řetězec v průběhu vývoje grafických karet zobecňoval. Z dříve pevně dané posloupnosti operací se stal programovatelný řetězec, jehož určité části lze dnes zaměnit za malý program zvaný shader. Shader se spouští pro každý prvek zvlášť, přičemž typ prvku přímo závisí na typu shaderu. Například vertex shader zpracovává vrcholy a fragment shader fragmenty obrazu. [3]

2.2 Způsoby programování řetězce

Při nahrazení pevně daných částí grafické karty programovatelnými procesory bylo potřeba vyvinout jednotný způsob pro definici vlastních shaderů. Samotné OpenGL nabízí pro vývojáře tři možnosti, jak s grafickou kartou pracovat. Jsou jimi jazyky ARB assembly, GLSL a Cg [4]. Další možnost pro tvorbu shaderů vyvinula společnost

Microsoft jako součást své technologie **Direct3D**. Jedná se o jazyk HLSL, který je proprietární obdobou jazyka GLSL.

Jelikož cílem práce je vyvinout nástroj pro tvorbu shaderů, je důležité zhodnotit, zda by možnost práce s více různými jazyky nepřinesla větší využitelnost editoru.

2.2.1 ARB assembly language

Tento jazyk je součástí rozšíření **ARB_vertex_program** a **ARB_fragment_program**. Nachází se na úrovni jazyků symbolických adres, ale nejde o čistou formu takového jazyka, přestože jim je velmi podobný. Oproti ostatním zmíněným jazykům je jeho kompilace velmi rychlá, což je způsobeno jeho extrémní jednoduchostí. Naopak jeho nevýhodou je omezení pouze na vertex a fragment shadery (jelikož jazyk vznikl ještě před GLSL 1.0) a přílišná složitost vzniklých programů v případě rozsáhlejších shaderů. [4]

2.2.2 OpenGL Shading Language

GLSL je programovací jazyk pro psaní shaderů, který vznikl jako součást standardu OpenGL. Jeho syntax je podmnožinou jazyka C, doplněný o specifické části použitelné v shaderech. Součástí je rovněž sada zabudovaných typů a funkcí, které budou popsány později v této práci.

Jazyk GLSL se vyvíjí souběžně s normou OpenGL a má i několik vzájemně nekompatibilních verzí. Verze jazyka korespondující s normou je popsána v tabulce 1.

Verze OpenGL	Verze GLSL
2.0	1.10
2.1	1.20
3.0	1.30
3.1	1.40
3.2	1.50

Tabulka 1: Verzování jazyka GLSL

Od verze OpenGL 3.3 je číslo verze jazyka GLSL shodné s verzí OpenGL. Významné milníky jsou především verze 1.50, kdy došlo k přidání geometry shaderů, verze 4.0, u které byly přidány tesselační shadery a verze 4.3, přinášející výpočetní shadery. [5]

2.2.3 C for Graphics

Tento jazyk byl společností nVidia navržen jako výchozí pro tvorbu shaderů. Sdružení ARB jej ale odmítlo a přiklonilo se k jazyku GLSL. Jazyk Cg je oproti GLSL více podobný jazyku HLSL (dokonce je možné portovat shadery z Cg do HLSL bez nutných změn, což jej činí multiplatformním) a je možné ho předkompilovat do finální spustitelné podoby již během tvorby programu. [4]

2.2.4 High Level Shader Language

HLSL je vysokoúrovňový programovací jazyk pro psaní shaderů, který vyvinula společnost Microsoft jako součást Direct3D. Jazyk je možné předkompilovat během tvorby programu nebo přenášet shader přímo v jazyce HLSL a kompilovat ho na místě. V rámci technologie Direct3D je rozlišováno několik takzvaných **shader modelů**, jak popisuje tabulka 2. [6]

Shader model je pevně daná sestava vlastností grafické karty. Aby grafická karta splňovala daný shader model, musí obsahovat všechny vyjmenované vlastnosti. OpenGL toto řazení nepoužívá, místo toho je možné dotázat se na dostupnost jednotlivých rozšíření pomocí funkcí knihovny. [7]

Verze modelu	Verze Direct3D	Verze GLSL	Vlastnosti
Shader Model 1			První omezená verze.
Shader Model 2	Direct3D 9	GLSL 1.30	
Shader Model 3			
Shader Model 4	Direct3D 10	GLSL 3.30	Přidán geometry shader.
Shader Model 5	Direct3D 11	GLSL 4.30	Přidány tesselační shadery.

Tabulka 2: Seznam shader modelů v Direct3D

Ve srovnání s jazykem GLSL nepřináší jazyk HLSL žádné významné výhody ani nevýhody. Existuje i způsob (například s využitím technologie TOGL od společnosti Valve [8]), jak přeložit shader z jazyka HLSL do GLSL. Proto nebude editace shaderů v jazyce HLSL součástí vznikajícího editoru.

2.3 Existující možnosti v editaci shaderů

Jako u většiny programovacích jazyků, je i při editaci shaderů možné využít různé programy. Výběr editoru souvisí s požadavky na podpůrné funkce, jakými jsou zvýrazňování syntaxe, doplňování kódu, opravy chyb v kódu nebo možnosti náhledu vytvářeného shaderu.

2.3.1 Editory prostého textu

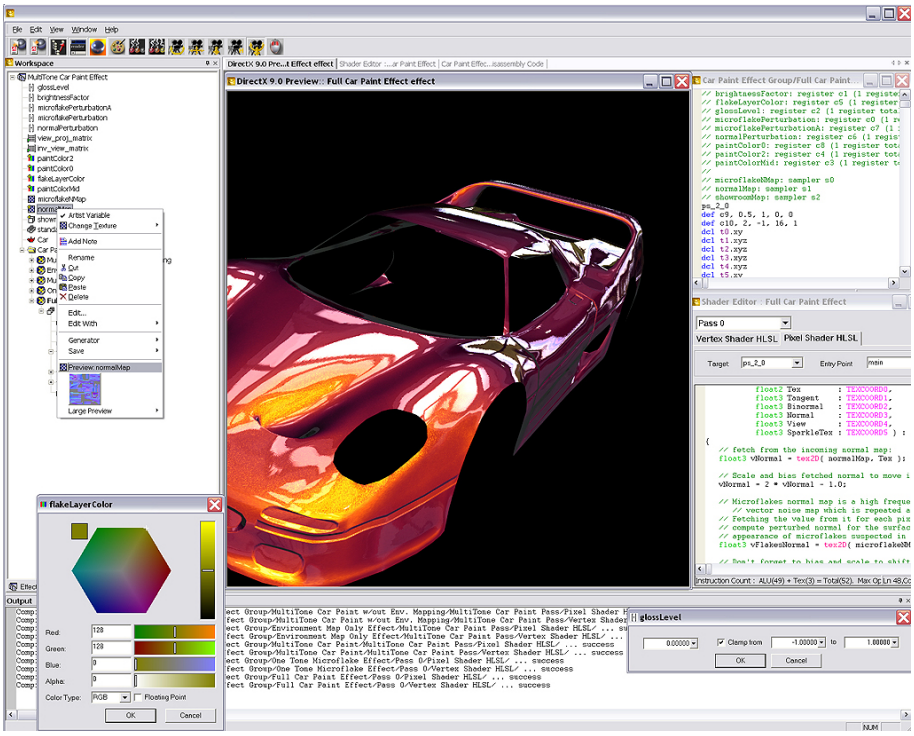
Protože shadery jsou svým rozsahem většinou velmi malé a čítají nejvýše desítky řádků kódu, první možnou volbou jsou klasické editory prostého textu. Na platformě Windows je typickým zástupcem této skupiny Poznámkový blok, pro linuxové systémy pak existují vizuální varianty jako gedit nebo Kate, případně editory pracující v CLI, jakými jsou nano, vi a další.

Poznámkový blok umožňuje pouze editaci textu, nerozpozná syntax jazyka ani nepodporuje žádnou formu jeho zvýraznění. Oproti tomu gedit podporuje zvýraznění syntaxe jazyka GLSL až do verze 4.0 [9], ale nezvládá doplňování kódu ani jakoukoliv podobnou funkci.

I přesto je psaní shaderů v editorech prostého textu poměrně časté, ale vyžaduje dobré znalosti jazyka i technologie. Editory tohoto typu neposkytují žádné možnosti doplňování ani ladění kódu, běžné z velkých specializovaných IDE. Absence náhledu z nich proto činí nástroj vhodný pouze pro znalce v oboru, pro výuku se nehodí.

2.3.2 RenderMonkey

Jedná se o editor vyvíjený společností AMD umožňující tvorbu shaderů pro programátory i designéry. Umožňuje zobrazovat shader v reálném čase nebo nastavovat vlastnosti shaderů (uniformní proměnné, modely, ...) skrze vizuální strom vlastností. Výhodou tohoto editoru je vedle vyjmenovaných vlastností i sada ukázkových shaderů a podpora Direct3D. Vývoj editoru byl ovšem ukončen, poslední verze je z roku 2008. Nepodporuje proto žádné později vydané rozšíření OpenGL. [10]



Obrázek 3: Náhled editoru RenderMonkey [44]

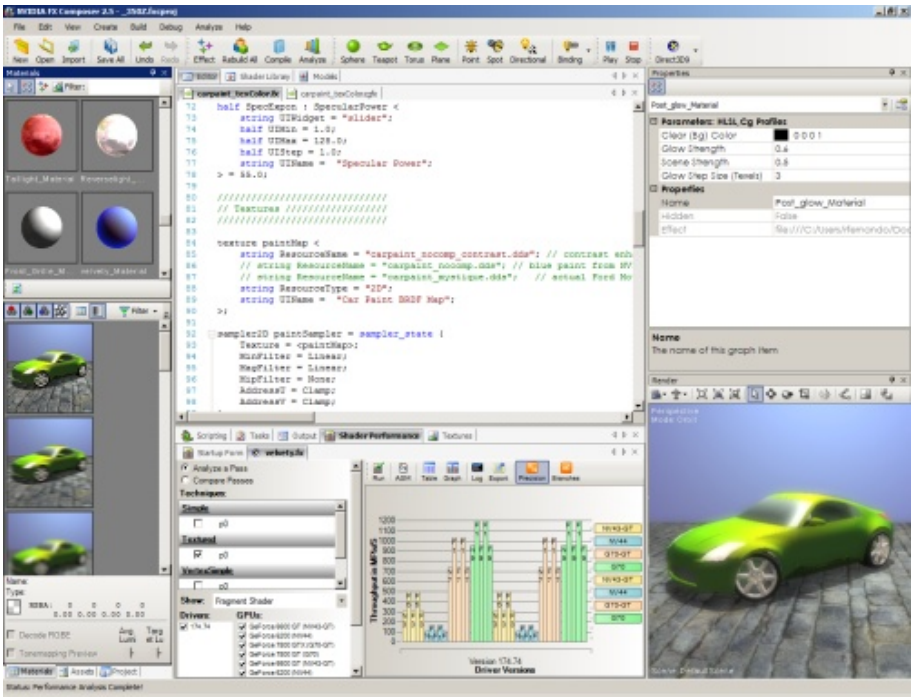
2.3.3 NVIDIA Nsight Development Platform

Platforma pro vývoj aplikací na CPU a GPU vyvinutá společností nVidia. Existuje jako rozšíření pro nástroje Microsoft Visual Studio (Windows) nebo Eclipse (Linux a Mac OS) [11]. Hlavní nevýhodou tohoto nástroje je úzká provázanost s grafickými kartami nVidia (vyžaduje konkrétní ovladače) a technologií CUDA. Ladění shaderů

podporuje pouze verze pro Visual Studio, a to jen pro jazyk HLSL. Rozšíření podpory pro GLSL a další vlastnosti se mají objevit v nastávající verzi 4.0, která je aktuálně ve stavu Release Candidate [12].

2.3.4 FX Composer

FX Composer je další nástroj pro vývoj shaderů od společnosti nVidia. Obsahuje podporu pro DirectX 10, včetně geometry shaderů, umožňuje vzdálené ladění přes protokol TCP/IP, nastavování vizuálních vlastností materiálů a ladění shaderů. Rovněž umožňuje zobrazit vytvářený shader na modelu, nahraném například z formátu COLLADA. Jeho vývoj byl ukončen verzí 2.5 v roce 2008. [13]



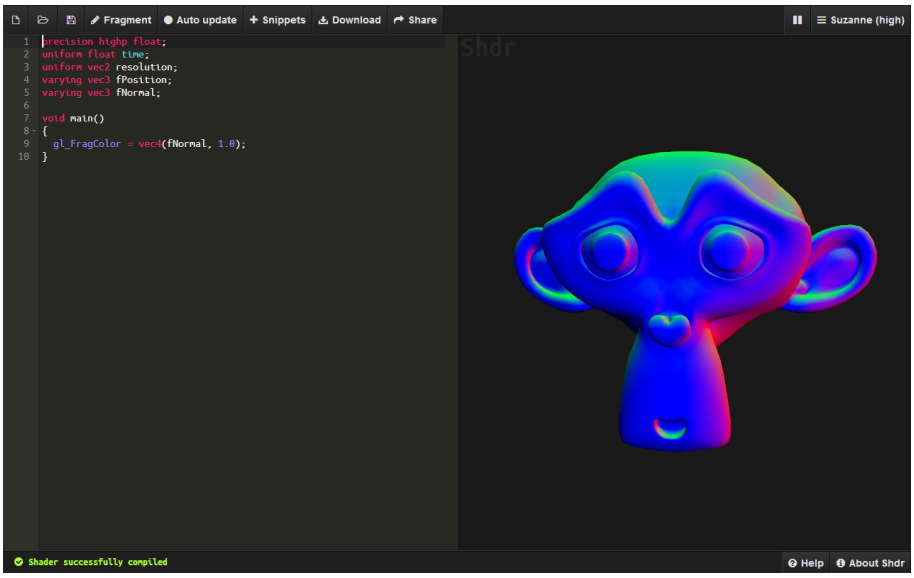
Obrázek 4: Náhled editoru FX Composer [45]

2.3.5 GPU PerfStudio 2

Nástroj vyvinutý společností AMD pro vývoj shaderů na platformě Windows. Podporuje Direct3D ve verzích 9, 10 i 11 a OpenGL 4.0. Umožňuje ladit shadery včetně zobrazení obsahu registrů, breakpointů apod. Je možné ho využít i na kartách společnosti nVidia. [14]

2.3.6 Online řešení

S příchodem WebGL se objevilo i několik možností, jak editovat shadery přímo ve webovém prostředí. Jedním z takových webových editorů je například Shdr Editor, dostupný na adrese `http://shdr.bkcore.com/`. Jasnou předností je kompilace a zobrazování shaderů v reálném čase. Ideálně se tedy hodí na výuku nebo experimentování. Nevýhodou je omezení OpenGL a GLSL pouze na vlastnosti, podporované normou WebGL.



Obrázek 5: Náhled editoru Shdr

Dalším zástupcem webových editorů je Shader Hub, sloužící současně jako úložiště shaderů. Na adrese `http://shaderhub.com/` lze nalézt postup, jak se Shader Hubem pracovat. Nástroj obsahuje pluginy pro Blender a další programy pracující s 3D modely. Jediná omezení vyplývají opět pouze z nutnosti použití WebGL.

V případě webových editorů se tedy nejedná o plnohodnotné shadery, a proto se podobné nástroje na vývoj profesionálních shaderů příliš nehodí.

3 OpenGL Shading Language

V následující části práce je popsán jazyk GLSL, jeho syntaktická pravidla a změny, kterými v průběhu svého vývoje prošel. Přestože vzniklý editor shaderů podporuje všechny verze tohoto jazyka, při jeho popisu je upřednostňována nejvyšší verze, pokud je to možné. Vycházeno bylo ze specifikace GLSL verze 4.4 [15].

Jazyk GLSL vychází syntakticky z jazyka C (není však jeho podmnožinou), z kterého přebírá většinu významných rysů. Jazyk C byl vybrán pro svou oblíbenost a rozšířenost v odborné komunitě, což nevyžaduje učit se pro použití jazyka GLSL nové příkazy a řídící struktury.

Shadery v jazyce GLSL jsou přenášeny v podobě zdrojových kódů. Nedochází tedy ke kompilaci v žádnou formu spustitelného kódu ani mezikódu, ale výsledný program se kompiluje až na místě použití. Jako kódování textu je použito UTF-8, ale po průchodu preprocesorem jsou z proudu znaků odebrány všechny sekvence, vyjma následujících:

- Písmena (`a-z`), (`A-Z`) a znak (_).
- Číslice (`0-9`).
- Symboly: (`.`), (`+`), (`-`), (`/`), (`*`), (`%`), (`<`), (`>`), (`[`), (`]`), (`(`), (`)`), (`{`), (`}`), (`^`), (`|`), (`&`), (`~`), (`=`), (`!`), (`:`), (`;`), (`,`) a (`?`).

Dále jsou ze zdrojového kódu odstraněny všechny komentáře (jsou nahrazeny znakem mezery). Podporovány jsou typy komentářů // a /* ... */, známé z jazyka C. Případně jsou odstraněny také konce řádků, což ale nemá vliv na hlášení o případných chybách v kódu. Speciální význam má v tomto případě znak (\), který je schopen potlačit konec řádku, pokud se nachází bezprostředně před ním. Z pohledu překladače se pak kód jeví tak, jak je popsáno ve zdrojovém kódu 1.

```
// zápis v původním zdrojovém kódu
float ci\
slo;

// kód po průchodu preprocesorem
float cislo;
```

Výpis 1: Význam opačného lomítka v GLSL

3.1 Preprocesor

Vedle již zmíněných vlastnostní preprocesoru, podporuje GLSL také makra, rozšířené o schopnosti maker z jazyka C++ a několik vlastních prvků. Seznam všech podporovaných direktiv je popsán ve zdrojovém kódu 2.

```
#
#define
#undef
```

```
#if
#ifdef
#ifndef
#else
#elif
#endif

#error
#pragma

#extension
#version

#line
```

Výpis 2: Seznam direktiv preprocesoru GLSL

Při podmiňování je možné použít operátor defined, tak jak je používán v jazyce C++. I ostatní vlastnosti převzatých maker zůstaly v jazyce GLSL zachovány. V makrech je dále možné použít speciální proměnné:

- __LINE__ pro číslo řádku.
- __FILE__ pro číslo zdrojového řetězce, který je právě zpracováván.
- __VERSION__ pro číslo verze jazyka GLSL, například 130 pro verzi 1.30.

Novinkou jsou makra #extension, #version a #line. Makro #line slouží k upravení číslování řádků. Za jménem makra se musí nacházet jedno nebo dvě celá čísla, oddělená mezerou. První číslo reprezentuje nové číslo řádku, druhé číslo pak nové číslo právě překládaného zdrojového kódu.

Význam maker #extension a #version je popsán v samostatných kapitolách níže.

3.1.1 Makro #version

Toto makro slouží k označení verze jazyka, pro který je určen zdrojový kód, na jehož začátku se makro nachází. Paramtery makra mohou být dva a oddělují se mezerami, přičemž pouze první z nich je povinný.

První argument určuje verzi jazyka GLSL tak, jak popisuje proměnná __VERSION__. Druhý nepovinný argument určuje profil jazyka GLSL. Specifikace definuje následující profily:

- core pro základní verzi OpenGL.
- compatibility pro zpětnou kompatibilitu mezi verzemi.
- es pro OpenGL ES u mobilních zařízení.

3.1.2 Makro #extension

Makro slouží k řízení rozšíření technologie OpenGL. Ve výchozím stavu překladače jsou všechna rozšíření vypnuta, což odpovídá použití makra `#extension all : disable`.

Jako parametr makra se uvádí název rozšíření (případně klíčové slovo **all** definující, že makro bude platit pro všechna dostupná rozšíření) následovaný dvojtečkou a způsobem chování, popsaným v tabulce 3.

Chování	Význam
require	Určuje, že dané rozšíření je potřebné k chodu tohoto shaderu. Pokud požadované rozšíření není dostupné nebo bylo jako název použito klíčové slovo **all**, způsobí vyvolání chyby, stejně jako makro #error.
enable	Povolí požadované rozšíření nebo v případě, že není dostupné, vyvolá **varování**. V případě, že bylo jako název použito klíčové slovo **all**, způsobí vyvolání chyby, stejně jako makro #error.
warn	Povolí požadované rozšíření a vyvolá **varování**. Klíčové slovo **all** způsobí **varování** na všech detekovatelných rozšířeních. V případě, že požadované rozšíření není rozpoznáno, způsobí vyvolání **varování**.
disable	Způsobí potlačení všech rozšíření, jakoby se v jazyce nenacházela. Pokud se v jazyce nachází kód, který dané rozšíření potřebuje, dojde k vyvolání **chyby** nebo **varování** tak, jak by se stalo v případě, že OpenGL dané rozšíření vůbec neobsahuje. Klíčové slovo **all** působí na všechna aktivní rozšíření. V případě, že požadované rozšíření není rozpoznáno, způsobí vyvolání **varování**.

Tabulka 3: Způsoby chování rozšíření v GLSL

3.2 Identifikátory a datové typy

Identifikátorem v jazyce GLSL se rozumí posloupnost znaků, odpovídající následujícím pravidlům:

- Identifikátor začíná na písmeno (pouze ASCII) nebo znak (_).
- Další znaky v identifikátoru mohou být i číslice (opět pouze ASCII).
- Identifikátory začínající na gl_ jsou rezervované.

Jako identifikátor proměnné/funkce není dále možné použít žádné, jazykem definované klíčové nebo rezervované slovo. Seznam takových slov se z důvodů jeho rozsáhlosti v práci nenachází a je možné jej dohledat v oficiální specifikaci jazyka. Obecně

jazyk GLSL používá obdobná klíčová a rezervovaná slova jako jazyky C a C++. Mezi klíčová slova se řadí i zabudované datové typy jazyka GLSL.

Jazyk GLSL definuje skalární i složené datové typy, přičemž je možné nadefinovat si vlastní struktury, podobně jako v jazyce C. Zde popsané typy byly přesně specifikovány v GLSL verze 1.30, předchozími verzemi se zde zabývat nebudeme. Jazyk GLSL obsahuje i zabudované složené typy. Detailnější popis datových typů a prace s nimi je popsána v následující části práce.

3.2.1 Logická hodnota (bool)

Tento datový typ reprezentuje logickou hodnotu a nabývá pouze dvou stavů:

- Pravda, vyjádřena literálem **true**.
- Nepravda, vyjádřena literálem **false**.

Narozdíl od jazyků C a C++ není možné kombinovat typ **bool** s typem **int**. V tomto směru se tedy GLSL chová jako jazyk Java.

3.2.2 Celočíselná hodnota (int, uint)

Zástupci celočíselných datových typů v GLSL jsou typy **int** a **uint**. Rozsah těchto typů je definován pevně na 32 bitů.

Datový typ **int** je obdobou typu signed long **int** z jazyka C, vyjadřuje tedy celé číslo se znaménkem v rozsahu $-(2^{31})$ až $(2^{31}-1)$. Typ **uint** je naopak obdobou typu unsigned long **int** z jazyka C, jeho rozsah je tedy 0 až $(2^{32}-1)$.

Pro rozlišení celočíselných literálů bez znaménka se používá přípona u nebo U. Samotný zápis literálů je pak prakticky totožný se zápisem známým z jazyka C. Příklady možných způsobu zápisu celého čísla ukazuje zdrojový kód 3.

```
int a = 0xFFFFFFFF; // plných 32 bitů odpovídajících hodnotě −1
int b = 0xFFFFFFFFU; // CHYBA! nelze přiřadit zápis typu uint do proměnné typu int
uint c = 0xFFFFFFFF; // plných 32 bitů odpovídajících hodnotě 0xFFFFFFFF
uint d = 0xFFFFFFFFU; // plných 32 bitů odpovídajících hodnotě 0xFFFFFFFF
int e = -1; // zápis hodnoty 1 zpracovaný výrazem (operátor −) na zápornou hodnotu, tedy −1
uint f = -1u; // zápis 1u zpracovaný výrazem na zápornou hodnotu, tedy 0xFFFFFFFF
int g = 3000000000; // zápis zabírající 32 bitů, nastavující znaménkový bit, tedy −1294967296
int h = 0xA0000000; // 32 bitový zápis v hexadecimálním tvaru se znaménkem
int i = 5000000000; // CHYBA! přesahuje rozsah 32 bitů
int j = 0xFFFFFFFFF; // CHYBA! přesahuje rozsah 32 bitů
int k = 0x80000000; // hodnota 0x80000000, tedy −2147483648
int l = 2147483648; // hodnota nastaví znaménkový bit, tedy −2147483648
```

Výpis 3: Příklady zápisu celočíselných literálů

3.2.3 Hodnota s pohyblivou řádovou čárkou (float, double)

Shadery jazyka GLSL podporují jednoduchou a dvojitou přesnost podle normy IEEE 754. Jednoduchou přesnost reprezentuje datový typ **float** (32 bitů), dvojitou přesnost, přidanou do GLSL ve verzi 4, pak typ **double** (64 bitů). Rozsahy těchto proměnných jsou přesně definovány dříve zmíněnou normou.

Pro rozlišení literálů čísel s pohyblivou řádovou čárkou stačí, aby zápis obsahoval desetinnou tečku (nalevo nebo napravo od ní nemusí být žádná číslice, ne však na obou stranách současně). V případě vynechání znaku tečky je možné literál rozlišit příponou `f` nebo `F`, pro dvojitou přesnost pak `lf` nebo `LF`. Samotný zápis literálů je pak prakticky totožný se zápisem známým z jazyka C, včetně vědeckého zápisu s exponentem. Příklady možných způsobu zápisu čísla s pohyblivou řádovou čárkou ukazuje zdrojový kód 4.

```
float a = 1f; // číslo s jednoduchou přesností
float b = 1.; // číslo s jednoduchou přesností
float c = 1.5; // číslo s jednoduchou přesností
double d = 2lf; // číslo s dvojitou přesností
double e = 3.1f; // číslo s dvojitou přesností
double f = .41f; // číslo s dvojitou přesností
double g = 5.0LF; // číslo s dvojitou přesností
double h = 101; // CHYBA! nelze kombinovat int a float
```

Výpis 4: Příklady zápisu literálů čísel s pohyblivou řádovou čárkou

3.2.4 Vektory

Vektory v GLSL jsou homogenní datové typy, složené z dvou, tří nebo čtyř prvků jednoho z typů:

- **float**
- **double** (předpona `d`)
- **int** (předpona `i`)
- **uint** (předpona `u`)
- **bool** (předpona `b`)

Název datového typu se skládá ze slova `vec` za kterým následuje číslo `2`, `3` nebo `4`, určující počet složek vektoru. Základní vektor má složky typu **float**, pro ostatní typy se musí uvést jednopísmenná předpona zmíněná výše.

Ke složkám vektoru lze přistupovat dvěma způsoby. První způsob je ekvivalentní s přístupem k poli. Druhou možností je dívat se na vektor jako na strukturu s prvky `x`, `y`, `z` a `w`. Druhý způsob umožňuje více možností, jak je popsáno v tabulce 4.

Další zajímavou technikou přístupu ke složkám vektoru je tzv. **swizzling**. Ten umožňuje získat z vektoru jiný vektor kombinací jeho původních složek. Pro otočení pořadí složek vektoru je možné použít kód `v = v.wzyx`; nebo je možné získat i kratší nebo

[0]	[1]	[2]	[3]	Srovnání s přístupem v poli.
.x	.y	.z	.w	Pro vektory reprezentující body nebo normály.
.r	.g	.b	.a	Pro vektory reprezentující barvy.
.s	.t	.p	.q	Pro vektory reprezentující souřadnice textur.

Tabulka 4: Způsoby přístupu k vektoru v GLSL

delší variantu vektoru použitím kódu: `myVec4 = myVec2.xyxx;` V rámci tohoto přístupu nelze kombinovat nesouvisející názvy složek (například `rbxy`) ani přesáhnout maximální délku vektoru.

Hodnotu vektoru lze zapsat pomocí konstruktoru, jehož název je shodný s požadovaným datovým typem. Pro získání vektoru se složkami 5 a 7 lze použít kód `vec2 v = vec2(5.0, 7.0);` a stejným způsobem lze zkrátit nebo doplnit již existující vektor: `myVec4 = vec4(myVec2, 5.0, 7.0);`

Délku vektoru je možné získat pomocí volání funkce `myVec4.length()`, která vrací konstantní hodnotu typu **int**.

3.2.5 Matice

Podobně jako u vektoru je matice složena z většího počtu složek, ale narozdíl od něj má dva rozměry. Šířka (počet sloupců) i výška (počet řádků) matice může být v rozsahu 2 až 4 a celkový rozměr matice může být libovolnou kombinací šířky a výšky. Pokud jsou oba rozměry shodné, uvádí se jako jedno číslo. Například pro matici o šířce i výšce 4 je v GLSL datový typ **mat4**. Pokud se rozměry liší, uvádí se jako první šířka, a po znaku `x` výška, například **mat2x3**.

Dalším rozdílem oproti vektorům je možnost vytváření matic pouze na základě datových typů **float** a **double**. Pro datový typ **double** má matice v názvu datového typu přidánu předponu `d`.

K maticím se přistupuje jako k dvourozměrným polím, kdy první souřadnice označuje sloupec a druhá řádek (číslováno od nuly). Matice je možné vytvářet pomocí konstruktorů, obdobně jako u vektorů. Pokud obsahuje konstruktor matice jediné skalární číslo, vznikne jednotková matice násobená tímto číslem. V běžném případě obsahuje konstruktor matice tolik čísel, kolik má matice složek. Skupinu po sobě jdoucích čísel lze nahradit vektorem odpovídající délky. Složky se do konstruktoru zapisují po sloupcích.

Velikost matice je možné získat pomocí volání funkce `myVec4.length()`, která vrací konstantní hodnotu typu **int**.

3.2.6 Samplery

Jedná se o objekty reprezentující textury. Ty mohou být libovolného rozměru a typu. Tento datový typ obsahuje hodnotu, umožňující manipulovat s příslušnou texturou. V shaderech se nachází zabudovaná sada specifických funkcí, které se samplery umožňují pracovat.

3.2.7 Struktury

Práce se strukturami v jazyce GLSL opět vychází z jazyka C, ale přináší některá logická zjednodušení (například vypuštění klíčového slova **struct** během deklarace proměnné). Existují zabudované typy, založené na strukturách, a je možné definovat si i vlastní strukturu. Typy struktur beze jména nelze vytvářet ani kombinovat (zanořit strukturu do struktury). Ukázka deklarace vlastní struktury je zobrazena ve zdrojovém kódu 5.

```
struct My { float f; };

struct My2 {
  My my; // takto strukturu vnořit lze
  float x;
} myStruct1;

myStruct1.x = 5; // přístup k prvku struktury
```

Výpis 5: Vlastní typ struktury

3.2.8 Pole

Ze všech výše zmíněných typů je možné vytvořit pole. Přístup k prvkům pole je stejný jako v jazyce C, ale některé mechanismy (například vytvoření pole) jsou v GLSL jedinečné. Možnosti práce s polem jsou demonstrovány ve zdrojovém kódu 6.

```
float cisla[3];

uniform vec4 body[8];

light svetla[];
const int pocet = 2;
light svetla[pocet];

// následující dva řádky jsou ekvivalentní
float a[5] = float[5](3.4, 4.2, 5.0, 5.2, 1.1);
float a[5] = float[](3.4, 4.2, 5.0, 5.2, 1.1);

// různé typy polí se stejným významem
vec4 a[3][2];
vec4[2] a[3];
vec4[3][2] a;

// zjištění délky pole
a.length() // 3
a[x].length() // 2
```

Výpis 6: Možnosti práce s polem v GLSL

3.3 Řídící struktury jazyka

V jazyce GLSL se nachází všechny známé konstrukce jazyka C, jako jsou podmínky, cykly a funkce. Novinkou je pouze operace **discard**. Krátké shrnutí nabízí zdrojový kód 7.

```
void funkce(float parametr, int dalsi)
{
  return;
}

void main()
{
  int x = 0;

  if(x == 5)
  {
  }
  else
  {
  }

  switch(x)
  {
  case 0:
    break;
  default:
  }

  for(int i = 0; i < 4; i++)
  {
  }

  while(x > 5)
  {
    continue;
  }

  do
  {
    continue;
  }
  while(x > 5);

  discard; // nový druh ukončení pouze pro fragment shadery
}
```

Výpis 7: Základní konstrukce jazyka GLSL

Platnosti proměnných a podobné prvky jsou totožné s jazykem C.

3.4 Výrazy a operátory

V jazyce GLSL se můžeme setkat se všemi známými operátory jazyka C. Jedinou významnější změnou je možnost kombinovat ve výrazech skaláry, vektory a matice, které

se v jazyce C nenacházely. Bližší popis fungování takových výrazů je popsán zdrojovým kódem 8.

```
mat3 m;
vec3 v, u, w;
float f;

// následující dva bloky kódu vykonají totéž
v = u + f;

v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;

// následující dva bloky kódu vykonají totéž
w = v + u;

w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;

// následující dva bloky kódu vykonají totéž
u = v * m;

u.x = dot(v, m[0]);
u.y = dot(v, m[1]);
u.z = dot(v, m[2]);

// následující dva bloky kódu vykonají totéž
u = m * v;

u.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z;
u.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z;
u.z = m[0].z * v.x + m[1].z * v.y + m[2].z * v.z;
```

Výpis 8: Výrazy s vektory a maticemi

3.5 Vestavěné funkce

Kompletní seznam všech vestavěných funkcích a jejich chování se nalézá ve specifikaci jazyka. V této kapitole je uveden pouze stručný výčet několika z nich.

Pro práci s úhly obsahuje GLSL funkce **radians** a **degrees**, které umožňují konverzi úhlových jednotek. Dále jsou dostupné všechny goniometrické funkce jako **sin**, **cos**, **tan** a jejich opačné nebo hyperbolické varianty.

Pro zjednodušení výrazů jsou zde obsaženy funkce, které jsou v jazyce C součástí hlavičkového souboru math, jako **pow**, **exp**, **log**, **sqrt**, **abs**, **round**, **ceil**, **floor**, **min**, **max** a další.

S ohledem na funkcionalitu shaderů je zde také sada geometrických funkcí:

- **length** přijímá vektor a vrací jeho délku

- **distance** přijímá dva vektory a vrací vzdálenost mezi nimi
- **dot** přijímá dva vektory a vrací jejich skalární součin
- **cross** přijímá dva vektory a vrací jejich vektorový součin
- **normalize** vrátí vektor téhož směru, ale s jednotkovou délkou

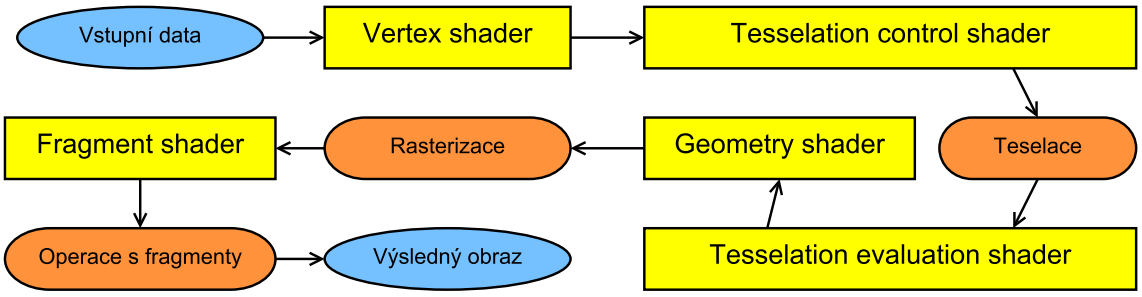
Další významnou skupinou funkcí jsou funkce pro práci s maticemi. Funkce **transpose** vrátí transponovanou matici, funkce **inverse** pak její invertovanou formu a pomocí funkce **determinant** lze vypočítat determinant matice.

Poslední zde popsanou skupinou jsou funkce pro manipulaci s texturami. Klíčovou roli zde hraje funkce **texture** (nahradila dříve používanou funkci **texture2D**), která v parametrech přijímá sampler a texturové souřadnice, přičemž vrátí příslušný texel ve formě vektoru.

3.6 Druhy shaderů

V poslední dostupné verzi jazyka GLSL (dostupné v době vzniku této práce) je možné pracovat s šesti typy shaderů:

- Vertex Shader
- Tesselation Control Shader
- Tesselation Evaluation Shader
- Geometry Shader
- Fragment Shader
- Compute Shader



Obrázek 6: Průchod dat programovatelným řetězcem

Jednotlivé typy shaderů jsou uvedeny tak, jak jimi prochází data skrze programovatelný řetězec (výjimkou je pouze Compute Shader, který neslouží k vykreslování obrazu). Průchod dat skrze shadery je zobrazen na obrázku 6.

3.6.1 Vertex Shader

Tento typ shaderů zpracovává vstupující vrcholy a jim příslušné vlastnosti. Data vrcholu zadává sám programátor a po průchodu vertex shaderem putují skrz řetězec do dalších shaderů. V rámci tohoto shaderu se provádí transformační operace s vrcholy a je možné si zde připravit data potřebná pro výpočet osvětlení. Vertex shader je v programovatelném řetězci přítomen od jeho první verze, avšak syntax jeho zápisu v GLSL se mezi jednotlivými verzemi měnil, jak ukazuje zdrojový kód 9.

```
// původní vertex shader v OpenGL 2.0
void main()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

// ve verzi OpenGL 3.1 byl již přepracován systém vstupů a matic
#version 140

uniform mat4 matice;

in vec4 bod;

void main()
{
  gl_Position = matice * bod;
}
```

Výpis 9: Ukázka vertex shaderu[3, 16]

3.6.2 Tesselation Control/Evaluation Shader

Možnost teselačních shaderů byla do OpenGL přidána ve verzi 4.0. První typ shaderu, tzv. tesselation control, slouží k přípravě dat pro teselátor grafické karty. Po průchodu teselátorem jsou data odeslána do druhého typu shaderu, tzv. tesselation evaluation, který přijme vstupní vrcholy a teselační koordinát a určí výsledné souřadnice vrcholu.

3.6.3 Geometry Shader

Geometry shader byl do OpenGL oficiálně přidán ve verzi 3.2 (do té doby byl dostupný jen jako rozšíření). Tento typ shaderu přijímá jako vstup všechny vrcholy daného primitiva a jeho výstupem je rovněž celé primitivum.

3.6.4 Fragment Shader

Spolu s vertex shaderem tvoří základní dvojci shaderů, které jsou v OpenGL od verze 2.0. Fragment shader zpracovává každý fragment obrazu a určuje jeho výslednou barvu. Základní zdrojový kód fragment shaderu je zobrazen v 10.

```
// původní fragment shader v OpenGL 2.0
void main()
{
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

// ve verzi OpenGL 3.1 je pak nutné definovat výstupní proměnnou ručně
#version 140

out vec4 vystup;

void main()
{
  vystup = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Výpis 10: Ukázka fragment shaderu[3, 16]

4 Návrh editoru

Před samotným započetím práce bylo potřeba zvolit vhodnou technologii pro vývoj editoru shaderů GLSL, navrhnout strukturu programu, rozložení grafického uživatelského rozhranní a celkovou funkcnionalitu programu.

4.1 Technologie

Protože shadery jazyka GLSL i samotná knihovna OpenGL jsou navrženy tak, aby bylo možné jejich použití na různých typech zařízení a platforem, je možné k implementaci editoru shaderů použít prakticky jakýkoliv programovací jazyk a sadu komponent. V rámci této práce bylo prozkoumáno několik možností přístupu k vývoji nově vznikajícího editoru a zjištěné poznatky byly shrnuty v této kapitole.

4.1.1 Programovací jazyk

Důležitým krokem pro projekt Shader editoru bylo vybrat programovací jazyk. Volba jazyka závisí na mnoha faktorech. Projekt by měl být použitelný na platformě Microsoft Windows, různých distribucích systému Linux a případně by měla být možná portace pro Mac OS X. Multiplatformost však není hlavní prioritou tohoto projektu. Další důležitou vlastností zvoleného jazyka je, aby obsahoval binding knihovny OpenGL, ideálně nativně (na nízké úrovni).

S ohledem na výše uvedené požadavky byl výběr omezen na jazyky, popsané v následující části práce.

Jazyk C Velmi oblíbený a dodnes používaný je **jazyk C**. Byl vyvinut v 70. letech 20. století Dennisem Ritchiem, aby se zjednodušil vývoj operačního systému Unix. Dnes je tento jazyk široce podporován na různých platformách a existuje k němu velké množství knihoven a nástrojů, což lze považovat za jeho velkou výhodu. Naopak nevýhodou je jeho stáří a s tím související absence moderních principů v programování. Jazyk v základu neobsahuje automatickou správu paměti, reflexi, apod. Přesto je dodnes rozšiřován (standardy C90, C99, ...), ale většina překladačů později přidané vlastnosti jazyka C nedokáže přeložit (částečně nebo úplně). [17]

Jazyk C++ Symbolickým nástupcem jazyka C (jazyk C není podmnožinou) je **jazyk C++**, který je objektově orientovaný. Stejně jako jeho předchůdce je široce podporován a těší se velké oblibě mezi programátory a odbornou veřejností. Byl vyvinut v Bellových laboratořích Bjarnem Stroustrupem začátkem 80. let 20. století. Ani tento jazyk neobsahuje automatickou správu paměti, reflexi a další moderní mechanismy, přestože se je snaha je v nových verzích jazyka přidat nebo nahradit obdobnými prvky. Syntax jazyka může působit nepřehledně a v jeho syntaktických pravidlech je s ohledem na jejich počet celkem složité se orientovat. Z tohoto důvodu bylo C++ umístěno na symbolické poslední místo. [18]

Jazyk Java je rovněž objektově orientovaný a spustitelný na většině současných platforem. Vyvinut byl Jamesem Goslingem ve společnosti Sun Microsystems, která ho poprvé představila v roce 1995. Dne 8. května 2007 došlo k uvolnění zdrojových kódů Javy (cca 2,5 miliónů řádků kódu) a jazyk se nadále vyvíjí jako open source. Na rozdíl od jazyků C a C++ se Java nepřekládá do nativně spustitelného kódu, ale využívá tzv. virtuální stroj ke svému běhu. Výhodou Javy je, že jako platforma v základu obsahuje všechny potřebné prvky, jako jsou kolekce, pokročilá práce se vstupy a výstupy nebo grafické uživatelské rozhranní (AWT, Swing). Nevýhoda Javy spočívá v absenci přímého bindingu knihovny OpenGL. Přesto lze s OpenGL pracovat za použití knihoven JOGL nebo LWJGL. [19, 20]

Jazyk Python Zástupcem interpretovaných jazyků vhodných pro implementaci této práce je jazyk **Python**. Jazyk navrhl Guido van Rossum v roce 1991. Přestože je jazyk využíván především pro skriptování, hodí se i k psaní celistvých aplikací (dobrým příkladem je 3D editor Blender). Python umožňuje vytvářet aplikace s grafickým uživatelským rozhranním pomocí přídavných knihoven, jakou je například Tkinter. Rovněž funkce OpenGL jsou dostupné pomocí rozšiřujících knihoven. Osobně nemám s vývojem v tomto jazyce mnoho zkušeností a jeho volbu pro tento projekt nepovažuji za přínosnou. [21]

Jazyk C# Dalším možným jazykem je jazyk **C#** vyvinutý společností Microsoft pro potřeby frameworku .NET v roce 2002. Jde o vysokoúrovňový objektově orientovaný jazyk vycházející z jazyků C a Java. Nevýhodou jazyka je jeho úzká provázanost s technologií .NET a tudíž i s operačním systémem Windows, přesto Microsoft označuje technologii .NET za otevřenou a multiplatformní. Implementace .NET pro unix-like systémy se jmenuje Mono a její vývoj je veden společností Novell. Mono přesto neposkytuje plnohodnotnou implementaci frameworku .NET, a proto není možná bezproblémová portace (například úplná absence Windows Presentation Foundation, která nebude podle aktuálních plánů realizována [22]). Výhodou je, stejně jako u Javy, že C# a .NET obsahují v základu grafické uživatelské rozhranní i potřebné knihovny. Stejně tak je nevýhodou neexistující přímý binding pro OpenGL. [23, 24]

4.1.2 Grafické rozhranní

Dále bylo nutné vybrat vhodnou technologii pro vytvoření grafického uživatelského rozhraní. Většina jazyků zmíněných v předchozí kapitole neobsahuje vlastní řešení. V současnosti je však dostupná řada nezávislých frameworků a knihoven, umožňujících tvorbu grafického prostředí.

Při volbě vhodné technologie je důležité, aby zvolené řešení obsahovalo všechny komponenty, části a funkce, které budou v době implementace editoru zapotřebí. Vybrané řešení by mělo být stabilní, odladěné a s dobrou dokumentací. S ohledem na dlouhodobý vývoj editoru by i vybraná technologie komponent měla mít předpoklad dalšího vývoje.

GTK+ Jedním z možných řešení bylo použití knihovny **GTK+**. Sada komponent GTK+ je napsána v jazyce C a existují bindingy do mnoha dalších jazyků. Je multiplatformní a šířena pod licencí LGPL, čili se pro potřeby projektu nabízí. [25]

Qt Další možností je knihovna **Qt**, kterou vyvinula společnost Trolltech v roce 1994. Knihovna je napsána v jazyce C++ a v případě komerčního užití má vlastní, proprietární licenci Qt Commercial Developer License. I pro tuto knihovnu existuje řada bindingů do jiných jazyků a je rovněž multiplatformní. [26]

wxWidgets Další knihovnu komponent, **wxWidgets**, začal vyvíjet Julian Smart v roce 1992 a její vývoj pokračuje dodnes. Knihovna wxWidgets poskytuje rozhraní pro vytváření multiplatformního grafického prostředí. Je vytvářena v jazyce C++, ale lze použít i v kombinaci s jinými jazyky. Nevýhodou této knihovny je právě její stáří, a s tím spojená absence modernějších přístupů. [27]

Swing a SWT V případě zvolení jazyka Java se nabízí sada komponent **Swing** (je součástí platformy Java SE), případně knihovna **SWT**. Swing je součástí desktopové Javy od verze 1.2 (rok 1997). Oproti tomu SWT začala vyvíjet společnost IBM jako výkonější náhradu za Swing. Na knihovně SWT je založena celá platforma Eclipse, takže je to ideální kandidát pro potřeby editoru. [28, 29]

XAML Pro technologii .NET byl vytvořen značkovací jazyk **XAML**. Jde o derivát jazyka XML vytvořený společností Microsoft jako součást Windows Presentation Foundation. XAML umožňuje oddělit rozložení a část funkcionality komponent mimo výkonný kód, psaný v jazyce C# (případně libovolném jazyce pro .NET framework). Nevýhodou XAMLu je neoddělitelnost od technologie .NET, a tedy i možnost použití pouze na operačním systému Windows. [30]

4.1.3 Shrnutí

Z výše popsaných možností byl zvolen pro implementaci shader editoru programovací jazyk C# se sadou komponent XAML. Protože ale technologie .NET (tedy i jazyk C#) neobsahuje propojení knihovny OpenGL, bylo nutné jej doplnit o kód v jazyce C.

4.2 Členění programu

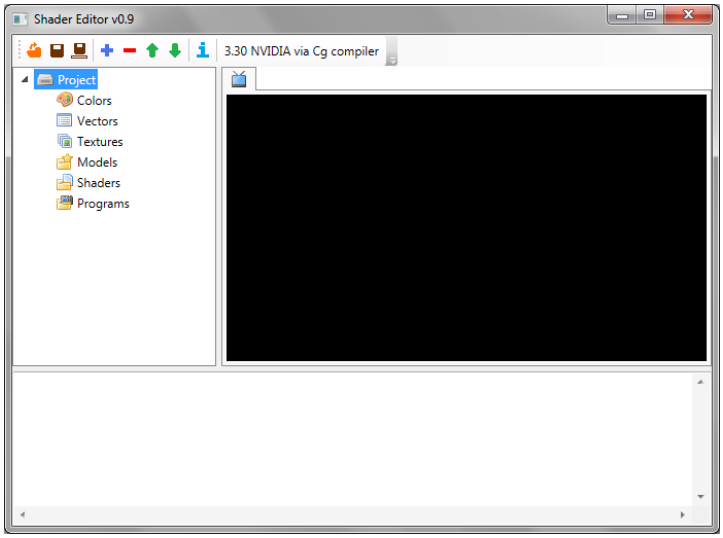
Při návrhu shader editoru bylo potřeba vyřešit několik klíčových otázek, od kterých se bude odvíjet výsledný postup vývoje celé aplikace. K tomuto účelu bylo prodeno několik testovacích implementací různých, na sobě nezávislých částí editoru.

4.2.1 Návrh fungování editoru

Ze získaných zkušeností s programováním a prací v jiných editorech shaderů byl vývoj editoru rozčleněn na několik základních částí. Z pohledu uživatele hraje nejvýznamější roli stromová struktura, která reprezentuje celý aktivní projekt, všechny jeho prvky a vztahy mezi nimi. Prvkem se z pohledu projektu rozumí:

- Textury
- Modely
- Proměnné
- Shadery
- Programy

Dalším klíčovou komponentou je panel náhledu, ve kterém se zobrazuje aktuálně rozpracovaná scéna. Samotná editace shaderu pak probíhá v panelu s textovým editorem, který dovede zvýraznit syntaxi shaderu a případně napovídat klíčová slova nebo jiné syntaktické prvky. Vše je doplněno panelem zobrazujícím výstup logovacích zpráv.



Obrázek 7: Rozložení komponent grafického rozhranní

4.2.2 Návrh implementace editoru

Jedním z možných způsobů implementace bylo, že celá výkonná část editoru bude naprogramována v jazyce C a grafické rozhranní napsané v jazyce C#/XAML bude sloužit pouze pro ovládání editoru (jako view v architektuře model-view-controller).

Pro tento případ bylo vyzkoušeno několik přístupů implementace stromu v jazyce C tak, aby umožnily programátorovi jednoduchou práci se shadery i editorem jako takovým. Žádný testovaný postup se však nejevil jako dostatečně vhodný, takže bylo rozhodnuto vyzkoušet jinou variantu přístupu.

Využito tedy bylo druhé možnosti, která spočívala v přenesení výkonné části do jazyka C#. Část editoru, napsaná v jazyce C, má tedy sloužit pouze jako podpůrná knihovna umožňující editoru jednoduchou práci s shadery a přístup ke knihovně OpenGL.

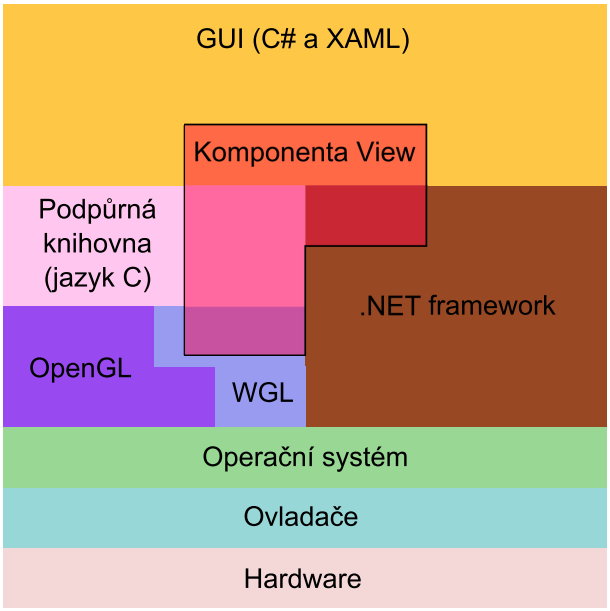
4.2.3 Viewport OpenGL v XAML

Stěžejním prvkem návrhu bylo vybrat řešení, jak v technologii komponent XAML zobrazit výsledek renderovaný pomocí OpenGL.

Jedním z existujících řešení je framework Tao, který rozšiřuje možnosti jazyka C# o prvky potřebné k vytváření počítačových her [32]. Méně robustním řešením by bylo použít projekt, jakým je například SharpGL, který pouze rozšiřuje C# o možnost práce s OpenGL [33]. S ohledem na nutnost přístupu ke všem funkcním knihovny OpenGL a potřebě pracovat vždy s nejnovějším dostupným API se žádné podobné řešení nejevilo dlouhodobě udržitelné.

V rámci této práce tedy bude vytvořeno vlastní propojení OpenGL a sady komponent XAML, které bude úzce souviset s podpůrnou knihovnou v jazyce C.

Znázornění celé struktury fungování editoru včetně komponenty s OpenGL zobrazuje obrázek 8.



Obrázek 8: Symbolické znázornění struktury editoru

4.2.4 Komunikace vrstev

Aby byla komunikace editoru v jazyce C# a podpůrné knihovny v jazyce C co nejjednodušší, byl návrh omezen na několik málo funkcí.

```
#define LIBEXPORT __declspec(dllexport)
#define LIBCALL __stdcall

typedef signed char Byte;
typedef signed long int Int;
typedef signed long long int Long;
typedef wchar_t Character;
typedef const Character* StringIn;
typedef Character* StringOut;
typedef void(__stdcall *Callback)(Int action, StringIn message);

LIBEXPORT Int LIBCALL dpCall(StringIn command, StringOut message);
LIBEXPORT Int LIBCALL dpPutData(Int id, Byte* data, Int length);
LIBEXPORT Int LIBCALL dpPutCallback(Callback callback);
LIBEXPORT Int LIBCALL dpPutHDC(Long pointer);
```

Výpis 11: Rozhranní v jazyce C

```
public const CharSet Encoding = CharSet.Unicode;

[UnmanagedFunctionPointer(CallingConvention.StdCall, CharSet = Encoding)]
public delegate void DPCallback(Int32 action, String message);

[DllImport(LibFileName, CharSet = Encoding, EntryPoint = "dpCall")]
public static extern Int32 Call(String command, StringBuilder message);

[DllImport(LibFileName, CharSet = Encoding, EntryPoint = "dpPutData")]
public static extern Int32 PutData(Int32 id, Byte[] data, Int32 length);

[DllImport(LibFileName, CharSet = Encoding, EntryPoint = "dpPutHDC")]
unsafe public static extern Int32 PutHDC(void* pointer);

[DllImport(LibFileName, CharSet = Encoding, EntryPoint = "dpPutCallback")]
public static extern Int32 PutCallback(DPCallback callback);
```

Výpis 12: Rozhranní v jazyce C#

Základ tvoří funkce Call, která umožňuje volání příkazů v pomocné knihovně. Příkazy jsou zadávány ve formě řetězců a výstupem volání je celé číslo, které nabývá hodnot 1 nebo 0 (1 pro správné vykonání a 0 pro chybu). Druhý parametr funkce Call slouží k vyzvednutí výsledku příkazu, pokud nějaký je. Pro vyzvednutí výsledku je nutné vyhradit minimálně 1024 znaků.

Druhou funkcí je PutData, která umožňuje předat knihovně velký blok dat. Prvním parametrem funkce je ID kontejneru, které je možné vytvořit pomocí příkazů. Ostatní již nesou samotná data. Význam návratové hodnoty je stejný jako u funkce Call.

Protože knihovna vykonává některé funkce asynchronně, potřebuje mít možnost kontaktovat editor. K tomuto účelu slouží funkce PutCallback, pomocí které lze nastavit

funkci pro zpětná volání. Funkce je pak knihovnou spuštěna pokaždé, kdy je potřeba aplikovat změnu. Prvním parametrem funkce zpětného volání je číslo akce, druhým řetězec nesoucí data. V aktuální implementaci je využívána pouze akce číslo 1, která reprezentuje zápis do logu.

Speciálním případem je pak funkce PutHDC, která je používána pouze na operačním systému Windows, sloužící k nastavení manipulátoru plátna pro potřeby propojení s OpenGL.

4.3 Formáty

Shader editor pracuje s externími zdroji uživatele, především pak s 3D modely a texturami. Program proto musí umět nahrát tyto data v různých, v praxi používaných formátech, z nichž některé jsou standardem.

4.3.1 Existující formáty textur

Textura je reprezentována polem texelů a může být jedno nebo vícerozměrná. V mnoha případech se setkáváme s dvourozměrnou reprezentací textury, v praxi uchovávanou jako obrázek či fotografii. Textury je možné načítat ze specializovaných formátů, nebo z formátů, které zná i běžný uživatel.

Formát JPEG Aplikace může přenášet texturu například ve formátů JPEG, který využívá ztrátovou kompresi a je primárně určen pro uložení fotografií. JPEG je otevřený formát s širokou možností podpory a většina grafických programů jej dokáže zpracovat. Jeho nevýhodou je, že je z výše uvedených důvodů omezen pro uložení pouze dvourozměrného obrazu. [31]

Formát PNG Dalším možným formátem je PNG, který byl vyvinut pro přenos obrazu po síti internet. Využívá bezztrátovou kompresi a na rozdíl od formátu JPEG umožňuje uchovat i alfa kanál. Nevýhodou je opět omezení na dvourozměrné obrázky. [31]

Formát TGA V praxi se můžeme hojně setkat i s formátem TGA, který je oproti předchozím dvěma výrazně jednodušší. Umožňuje surová data uložit přímo nebo případně využít jednoduché kódování. Kvůli své jednoduchosti ho využívá mnoho programů i her, a to i k vnitřním účelům. [31]

Formát DDS Žádný z výše jmenovaných ale nepodporuje vícerozměrné textury, ani kompresi přímo kompatibilní s grafickou kartou. Formátem, který tato kritéria splňuje, je například DirectDraw Surface. Vyvinut byl firmou Microsoft a může být využit v kombinaci s DirectX i OpenGL. Formát sám je dobře zdokumentovaný, ale využívá kompresi chráněnou patentem. [34]

4.3.2 Podporované formáty textur

Bylo potřeba vyřešit způsob načítání texturových/obrázkových formátů, zmíněných v kapitole 4.3.1. Pro standartní formáty obrázků, jako je JPEG, PNG a TGA, existují samostatně použitelné svobodné knihovny **libjpeg**, **libpng** a **libtga**. Dalším řešením bylo použití komplexní knihovny pro nahrávání obrázků, jakou je například knihovna **FreeImage**. Nevýhodou takového řešení je větší paměťová náročnost celé aplikace, výhodou pak možnost načítání nepřeberného množství formátů.

Seznam všech množných knihoven, které byly pro účely této práce zkoumány, je popsán v tabulce 5.

Název	Formáty	Web
libjpeg	JPEG	http://libjpeg.sourceforge.net/
libpng	PNG	http://www.libpng.org/
libtga	TGA	http://tgalib.sourceforge.net/
freeimage	JPEG, PNG, TGA, DDS	http://freeimage.sourceforge.net/
DevIL	JPEG, PNG, TGA, DDS	http://openil.sourceforge.net/

Tabulka 5: Knihovny pro načítání různých formátů obrázků

4.3.3 Specifikace formátu T1

Pro interní účely této práce byl vytvořen jednoduchý formát pro přenos textur, který je ekvivalentem formátu **M1** pro uložení modelů (kapitola 4.3.6). Formát je vhodný jak pro uložení textury do souboru, tak jako kontejner pro uchování nahraných dat, aby si je mohly jednotlivé vrstvy programu efektivně předávat. Při vytváření tohoto formátu byl kladen důraz na to, aby byla režie nutná pro jeho načtení i práci s ním co nejmenší. Přestože to nebylo podmínkou, byla zde snaha udržet formát optimální i pro mobilní platformy. Výsledek je proto kompatibilní s **OpenGL ES**, dovede uchovat pouze surová data (nepodporuje hardwarovou kompresi textur) a nepodporuje práci s čísly s plovoucí řádovou čárkou. Důvodem je udržení jednoduchosti.

Část	Velikost	Poznámky
Magické číslo	ushort	Číslo T1 s hodnotou `((84 « 8)	49)`
Typ uložení pixelů	ushort	Počet bitů podle masky `RRRR GGGG BBBB AAAA`.
Rozměr šířky	ushort	Platné jsou pouze mocniny dvou větší než 0.
Rozměr výšky	ushort	Platné jsou pouze mocniny dvou větší než 0.

Tabulka 6: Hlavička formátu T1

K načtení formátu je potřeba dvou datových typů: **byte** (jednobajtové číslo bez znaménka; v jazyce C *unsigned char*) a **ushort** (dvoubajtové číslo bez znaménka; v jazyce C *unsigned short*). Všechna data jsou uložena ve **velké endianitě**. Formát dovede uchovat pouze dvourozměrné textury. Hlavička formátu má 8 bajtů, jejichž význam popisuje tabulka 6.

Magické dvoubajtové číslo na začátku hlavičky slouží pro identifikaci formátu a současně jako ověření správného uložení bajtů (endianity). Následující **ushort** obsahuje formát uložení jednotlivých pixelů. Každé 4 bity reprezentují číslo v rozsahu 0-15 vyjadřující, kolik je použito bitů na každou složku barvy (RGBA, v tomto pořadí) zvlášť. Například pro formát R5-G5-B5-A1 by byla hodnota této části hlavičky rovna `0x5551` hexadecimálně. Typ uložení pixelů, určeného v hlavičce, musí vyhovovat jedné z možností z tabulky 7.

R	G	B	A	Suma	Formát	Typ
0	0	0	8	8	GL_ALPHA	GL_UNSIGNED_BYTE
8	8	8	0	24	GL_RGB	GL_UNSIGNED_BYTE
5	6	5	0	16	GL_RGB	GL_UNSIGNED_SHORT_5_6_5
8	8	8	8	32	GL_RGBA	GL_UNSIGNED_BYTE
4	4	4	4	16	GL_RGBA	GL_UNSIGNED_SHORT_4_4_4_4
5	5	5	1	16	GL_RGBA	GL_UNSIGNED_SHORT_5_5_5_1
8	0	0	0	8	GL_LUMINANCE	GL_UNSIGNED_BYTE
8	0	0	8	16	GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE

Tabulka 7: Formát uložení pixelů formátu T1

Dále se v hlavičce nachází dvě **ushort** hodnoty, reprezentující šířku a výšku textury. Rozměry musí být pouze mocniny dvou a nesmí být 0.

Za hlavičkou až do konce souboru se nachází samotná data textury. Ty je možné s ohledem na jejich formát načíst jako blok (pole) bajtů a předat přímo OpenGL. Data textury je možné vnímat jako pole bajtů délky *šířka * výška * jednotka*. Jednotka se vypočítá součtem všech složek pixelu a vydělení tohoto součtu číslem 8 (např.: pro formát 5-5-5-1 je součet 16 a po vydělení 8 výjde 2, tedy jeden pixel zabírá 2 bajty).

Důležité je, že pro formáty pixelů reprezentované jako typ **ushort**, musí být hlídána endianita. Typ lze vyčíst z tabulky 7 a sloupce **Typ**.

4.3.4 Existující formáty modelů

Modelem se v tomto případě rozumí změť trojúhelníků (případně jiných primitiv) nebo může jít i o část scény, včetně popisných informací. V editoru se modely využívají k demostraci funkcionality shaderů. V následující části jsou popsány nejčastěji používané formáty pro uložení modelů, které se pro účely této práce nejlépe hodí.

Wavefront OBJ je textový formát (kódování ASCII), který obsahuje příkazy psané po řádcích. Příkaz se skládá ze jména, což může být jedno nebo více písmen, následovaného parametry, nejčastěji ve formě čísel. Byl vyvinut pro potřeby aplikace Wavefront's Advanced Visualizer a dnes s ním dokáže pracovat mnoho grafických nástrojů. Samotný model je pak většinou rozložen do dvou souborů: geometrii obsahující soubor s příponou .obj a (většinou stejnojmenný) soubor s příponou .mtl, obsahující

informace o materiálech. Existuje i binární varianta tohoto formátu (používá příponu .mod), která je ale proprietární a nezdokumentovaná. [35]

COLLADA je výměnný formát pro 3D grafiku s širokou podporou. Je založen na formátu XML a používá příponu .dae nebo .zae pro zazipovanou variantu. Původně byl vyvinut společností Sony pro potřeby konzolí PlayStation 3 a PlayStation Portable. Od verze 1.4 jej spravuje skupina Khronos, stejně jako OpenGL. Formát byl v následujících verzích rozšířen i o podporu OpenGL ES a možnost shaderů. Vedle samotného modelu dovede formát nést informaci o celé scéně, včetně osvětlení, fyzikálních dat, chování prvků, kamer, a díky možnostem XML je možné jej rozšířit o jakoukoliv potřebnou schopnost. S ohledem na jeho komplexnost je jeho největší nevýhodou poměrně velká složitost a režie nutná k jeho načtení (zdrojové kódy implementace OpenCOLLADA mají téměř 100 MB a obsahující přes 2700 souborů[36]). [37]

4.3.5 Podporované formáty modelů

Pro první verzi editoru jsme zvolil formát Wavefront OBJ, pro jehož načtení bude použit jednoduchý parser vytvořený v rámci této práce. Do budoucna je plánována podpora dalších 3D formátů, ať už přímo zabudovaných v práci, nebo případně formou rozšířitelných modulů.

4.3.6 Specifikace formátu M1

Stejně jako v případě textur, i pro modely byl sestaven vlastní formát jménem M1. Formát uchovává vždy jeden statický 3D mesh a je navržen jako kontejner pro uchování nahraných dat pro předávání mezi vrstvami programu. Tak jako formát **T1**, i M1 nevyžaduje velkou režii pro načtení, a je optimální pro mobilní platformy. Formát neobsahuje žádnou podporu komprese.

K načtení formátu je potřeba dvou datových typů: **ushort** (dvoubajtové číslo bez znaménka; v jazyce C *unsigned short*) a **float** (číslo s pohyblivou řádovou čárkou s jednoduchou přesností podle IEEE 754 [38]; v jazyce C *float*). Všechna data jsou uložena ve **velké endianitě**. Hlavička formátu má 8 bajtů, jejichž význam popisuje tabulka 8.

Část	Velikost	Poznámky
Magické číslo	ushort	Číslo M1 s hodnotou `((77 « 8) | 49)`
Typ uložení vrcholu	ushort	Počet složek podle masky `TTTT CCCC NNNN VVVV`.
Počet vrcholů	ushort	Platné jsou pouze hodnoty větší než 0.
Počet indexů	ushort	Platné jsou všechny hodnoty včetně 0.

Tabulka 8: Hlavička formátu M1

Magické dvoubajtové číslo na začátku hlavičky slouží pro identifikaci formátu a současně jako ověření správného uložení bajtů (endianity). Následující **ushort** obsahuje formát uložení jednotlivých vrcholů. Každé 4 bity reprezentují číslo v rozsahu 0-15

vyjadřující, kolik je použito floatů na každou složku vrcholu (T, C, N a V, v tomto pořadí) zvlášť. Význam složek je následující:

- T - Texturové koordináty T1, T2, T3 nebo T4 ($S, [T, [P, [Q]]]$)
- C - Barvu vrcholu C3 nebo C4 ($R, G, B, [A]$)
- N - Normálu N3 (NX, NY, NZ)
- V - Souřadnice vrcholu V2, V3 nebo V4 ($X, Y, [Z, [W]]$)

Platné hodnoty pro složky jsou následující:

- Pro $T \in \{0, 1, 2, 3, 4\}$
- Pro $C \in \{0, 3, 4\}$
- Pro $N \in \{0, 3\}$
- Pro $V \in \{2, 3, 4\}$

Způsob jak vypočítat celkovou velikost vrcholu (počet floatů na vrchol) popisuje tabulka 9.

Formát	T	C	N	V	Počet floatů
V_a				x	a
T_aV_b	x			x	a+b
C_aV_b		x		x	a+b
N_aV_b			x	x	a+b
$T_aC_bV_c$	x	x		x	a+b+c
$T_aN_bV_c$	x		x	x	a+b+c
$C_aN_bV_c$		x	x	x	a+b+c
$T_aC_bN_cV_d$	x	x	x	x	a+b+c+d

Tabulka 9: Způsob výpočtu celkové velikosti vrcholu formátu M1

Dále se v hlavičce nachází dvě **ushort** hodnoty, reprezentující počty uložených vrcholů a indexů. Pokud je počet indexů roven 0, nejsou data indexů ve formátu uložena.

Za hlavičkou až do konce souboru se nachází samotná data modelu, nejdříve pole vrcholů a pak bezprostředně za ním (není zde žádný čitelný oddělovač) pole indexů. Obě skupiny dat je možné s ohledem na jejich formát načíst jako blok (pole) bajtů a předat přímo OpenGL.

Při nahrávání pole vrcholů je třeba dávat pozor na délku pole. Číslo uložené v hlavičce reprezentuje **počet vrcholů**, nikoliv počet prvků v poli. Pro zjištění délky pole je nutné vynásobit číslo z hlavičky celkovou velikostí vrcholu, jak popisuje tabulka 9.

Pole indexů má stejný počet prvků, jaký je uložen přímo v hlavičce. Speciálním případem je situace, kdy je v hlavičce číslo 0. V tomto případě není pole indexů v souboru

přítomno, ale je nutné ho vytořit jednoduchým postupem. Jeho délka je stejná jako počet vrcholů a jeho obsahem jsou hodnoty od 0 po $n - 1$ (kde n je délka pole).

Důležité je, že pro obě skupiny dat, ať už reprezentované jako typ **ushort** nebo **float**, musí být hlídána endianita.

4.3.7 Pomocné formáty V1 a P1

Pouze pro účely komunikace s podpůrnou knihovnou jsou v projektu obsaženy ještě dva kontejnery s názvy V1 a P1. Stejně jako T1 a M1 jsou přenášeny ve velké endianitě a sdílí s nimi mnoho rysů.

V1 slouží k přenosu vektorů a matic a má podobný tvar, jako formát T1. Na jeho začátku je **ushort** s identifikátorem V1, další **ushort** v pořadí má vždy hodnotu 0 a slouží k zarovnání s formátem T1. Třetí a čtvrtý **ushort** reprezentují šířku a výšku v rozsahu 1 až 4. Vektory pak mají výšku 1. Za hlavičkou následuje pole typu float o délce šířka krát výška.

P1 slouží k přenosu posloupnosti programu a názvů uniformů. Na začátku je opět **ushort** jenž identifikuje formát P1 ve formě ASCII. Vedle typu **ushort** a **byte** je k načtení P1 potřeba ještě datový typ **uint**, představující 32-bitové celé číslo bez znaménka. Po indentifikátoru formátu následuje **uint**, určující celkový počet záznamů. Poté již následují samotné záznamy. Záznam je reprezentován dvojicí **uint** a **byte**, kde **uint** představuje identifikátor elementu a **byte** délku pole bajtů, které (pokud je tato hodnota nenulová) bezprostředně následuje. Toto pole je vlastně řetězec jazyka C korektně zakončený znakem s hodnotou 0 a představuje název uniformu v kódování ASCII.

4.4 Projekt

Uživatel pracující s editorem bude mít možnost si rozdělanou práci uložit jako projekt a ten následně načíst. Projekt bude reprezentován souborem formátu JSON [39], který bude obsahovat všechna popisná data stromu projektu a odkazy na ostatní soubory (včetně jejich původních názvů). Na soubory se bude odkazovat jak relativně, tak absolutně, podle volby uživatele.

Další plánovanou možností bude uložení projektu do archivu formátu ZIP. Uvnitř archivu nebude žádná struktura složek, nebudou zachována původní jména souborů a rovněž komprese vzniklého ZIP souboru bude pouze volitelná.

Klíčovým souborem uvnitř archivu bude **project.json**, představující samotný projekt. Ostatní soubory v archivu budou mít název vždy **node-*ID***, kde ID je číslo nebo vnitřní identifikátor pro potřeby editoru.

5 Implementace

V této kapitole je popsán postup implementace celého shader editoru se speciálním zaměřením na zajímavé, problematické a klíčové části. Celá implementační fáze probíhala na operačním systému **Windows 7** za pomocí nástroje **Microsoft Visual Studio 2012**. Během vývoje byl celý projekt migrován pod nástroj **Microsoft Visual Studio 2013**, který nabízí vylepšenou podporu překladu jazyka C.

Implemenetace probíhala v několika fázích, odpovídajících jednotlivým částem programu. Jako první byla naplánována implementace komponenty **GLView**, která umožňuje využívat OpenGL v sadě komponent XAML. Následně byl vývoj zaměřen na podpůrnou knihovnu v jazyce C. Dalším krokem pak byl návrh grafického uživatelského rozhranní aplikace v jazyce XAML. Komponentám byla věnována i v další část implementace, kdy byla řešena stromová struktura projektu a řídící část aplikace, včetně nahrávání a konvertování různých formátů. Po vybrání vhodné komponenty ke zvýrazňování syntaxe byly řešeny pouze drobné úpravy projektu.

5.1 Komponenta GLView

Aby bylo možné zobrazit na operačním systému Windows výstup z OpenGL, je nutné využít rozhranní zvané **WGL**. Toto rozhranní umožňuje připojit obrazový výstup k plátnu komponenty, vytvořené pomocí **WinAPI**. K tomuto účelu je potřeba získat manipulátor, který je ve WinAPI reprezentován datovým typem **HWND**. Z něj je možné příslušnou funkcí získat manipulátor plátna, což je hodnota datového typu **HDC**. Prvním úkolem tedy bylo získat HDC komponenty, nacházející se v okně editoru. [3]

Technologie XAML byla představena s verzí .NET framewotku 3.0 jako součást Windows Presentation Foundation. Hlavním cílem tohoto snažení bylo vyvinout moderní sadu komponent, která by umožňila plně oddělit vzhled od funkcionality aplikace a nahradit tím, dnes již zastaralou, sadu komponent WinForms, rovněž z dílny společnosti Microsoft. XAML je tedy plně závislý na technologii .NET a není přímo provázán se starším WinForms a WinAPI. Z tohoto důvodu není možné získat manipulátor použitelný skrze WinAPI, protože XAML tuto technologii ke svému chodu nevyužívá, a pouze na ní hostuje. [41]

Oproti tomu komponenty sady WinForms rozhraní WinAPI používají a každá taková komponenta má tedy svůj vlastní manipulátor. Řešením se ukázalo být použití XAML komponenty `System.Windows.Forms.Integration.WindowsFormsHost`, která je součástí implementace XAML pro operační systém Windows a umožňuje XAMLu hostovat komponenty z prostředí WinForms.

5.1.1 Propojení s WinForms

Stačilo tedy vytvořit komponentu pro WinForms, která bude fungovat jako náhled výstupu OpenGL. Pro tento účel byla vytvořena třída `GLView`, která je potomkem třídy `System.Windows.Forms.Control`. Této komponentě bylo nutné nastavit příznaky `AllPaintingInWmPaint` a `UserPaint`.

Následně se získá instance třídy System.Drawing.Graphics, reprezentující plátno komponenty. Z ní je možné získat strukturu typu System.IntPtr, reprezentující manipulátor HDC, jehož hodnota je předána podpůrné knihovně pomocí funkce PutHDC.

Ošetřeny byly rovněž události překreslení a změny velikosti komponenty, při kterých je podpůrné knihovně zaslán příkaz `viewport repaint` *`šířka výška`*, který (v případě nutné změny velikosti) obsahuje rozměr komponenty. Současně je nutné po celou dobu běhu programu uchovat instanci plátna a pomocné struktury HDC, jinak může dojít k jejich uvolnění garbage collectorem a platné manipulátory WinAPI budou zrušeny. [42]

5.1.2 Řešené problémy

Během vytváření komponenty GLView bylo nutné vyřešit několik problémů.

HDC se získává ze struktury typu System.IntPtr a bylo nutné zajistit předání ukazatele do podpůrné knihovny (tedy jazyka C). V prvních verzích editoru bylo experimentováno s možností předávání ukazatelů v tzv. **unsafe** módu. Ten spočívá v použití klíčového slova unsafe (před metodou, blokem kódu apod.), které umožní vytvářet neověřovaný kód. Aby byl vzniklý kód platný, je nutné ve vlastnostech projektu zapnout příslušný příznak překladače. Přestože by toto řešení bylo funkční, bylo rozhodnuto, s ohledem na vyšší stabilitu, z programu neověřovaný kód vyloučit. [40]

Manipulátor HDC je tedy v editoru předáván jako 64-bitové celé číslo a jeho zpracování je řešeno až na úrovni jazyka C.

Kompilace části projektu, která je vytvářena v jazyce C#, je nastavena tak, aby byla upřednostněna 32-bitová architektura. Pokud je tato volba vypnutá, dojde při sputění projektu vyvolání výjimky typu System.Windows.Markup.XamlParseException. Na základě zkušeností lze odhadovat, že tento problém souvisí s kompilací podpůrné knihovny, která je za normálních okolností 32-bitová. Protože jedinou informaci, kterou ladící nástroj poskytne, je pozice ve zdrojovém kódu XAML a další krokování platforma .NET neumožňuje, nepodařilo se důvod výskytu této chyby potvrdit.

5.2 Podpůrná knihovna

Vedle části komponty GLView obsahuje podpůrná knihovna editoru také sadu příkazů, které umožňují na již zmíněnou komponentu vykreslovat obraz pomocí OpenGL. Samotná knihovna je napsána v jazyce C, konkrétně v dialektu normy C99. V rámci kódu je využíváno vlastností, které novější forma přináší. Současně je kód psán tak, aby splňoval syntaktická pravidla jazyka C++, v případě že by překladač jazyka C nebyl dostupný nebo nepodporoval výše zmíněné C99.

5.2.1 Ovládání a uložiště

Pro dosažení co nejnižších nároků na výkon jsou předpřipravená data, potřebná pro volání funkcí OpenGL, uchovávána na straně podpůrné knihovny. Práce s nimi je k dispozici přes sadu příkazů, které umožňují uložiště spravovat. Data jsou v uložišti strukturována do tzv. elementů, které jsou reprezentovány identifikátorem ve tvaru celého (kladného a nenulového) čísla.

Pro založení elementu slouží příkaz `create`. Celý tvar příkazu je `create` *`type id`*, kde *type* je druh vytvářeného elementu a *id* reprezentuje jeho identifikátor. Druhy elementů jsou následující:

- `vector`
- `color`
- `vertex shader`
- `tesselation control shader`
- `tesselation evaluation shader`
- `geometry shader`
- `fragment shader`
- `model`
- `texture`
- `program`

Jakmile je element vytvořen, je možné do něj pomocí funkce PutData vložit obsah, odvíjející se od druhu vytvořeného elementu. Pro elementy `vector` a `color` je nutné použít formát V1. Data všech druhů shaderů odpovídají zdrojovému kódu shaderu. Modely pak používají k přenosu formát M1, textury T1 a programy P1, blíže popsané v kapitole 4.3.

Změny jsou aplikovány ihned po vložení nových dat, není proto nutné volat žádné aktualizační příkazy apod.

5.2.2 Další příkazy

Vedle příkazů pro práci s uložištěm obsahuje knihovna ještě další pomocné přikazy. Důvodem je nutnost editoru přistupovat k nízkoúrovňovým funkcím knihovny OpenGL jednotným a snadným způsobem.

Pro zjištění informací o dostupné implementaci OpenGL slouží příkazy z kategorie `info`. Dostupnou verzi jazyka GLSL je možné získat voláním příkaz `info shader`. Získaná informace je vrácena skrze druhý parametr funkce Call.

Práci s plátnem umožňuje sada příkazů z kategorie `viewport`. Pro překreslení lze použít příkaz `viewport repaint`. Alternativně lze pomocí tohoto příkazu změnit velikost plátna zadáním šířky a výšky, čehož přímo využívá komponenta GLView.

5.3 Grafické uživatelské rozhraní

Rozložení komponent bylo již zmíněno v kapitole 4.2.1 na obrázku 7. Sada komponent XAML obsahuje v základu většinu komponent, potřebných pro vytvoření celého editoru. Panel nástrojů tvoří komponenta ToolBar, dokovaná v panelu ToolBarTray. Pracovní plocha je pak členěna pomocí TabControl, ve které je pevně ukotvená karta s komponentou GLView. Výstup logu je pak tvořen obyčejným TextBox. Zásadním prvkem celého rozločení je rozhodně komponenta TreeView, která představuje samotný projekt. Ta také prošla největší úpravou.

5.3.1 Komponenta TreeView

Během vývoje grafické části bylo potřeba vybrat vhodnou komponentu pro reprezentaci stromu projektu. Pro použití se nabízela komponenta TreeView, která je přímo součástí XAML, nebo některá z komponent třetích stran.

Nevýhodou vestavěné komponenty je její přílišná jednoduchost. TreeView v základu nepodporuje některé pro editor potřebné funkce, jako je drag&drop či pokročilá práce s uzly. Jelikož komponenta stromu projektu tvoří stěžejní část editoru a volba komponenty třetí strany by způsobila závislost na jejím dalším vývoji. Z tohoto důvodu bylo věnováno úsilí do rozšíření možností stávající komponenty TreeView.

Pro potřeby editoru byla komponenta rozšířena o možnost přetahování uzlů za pomoci myši, přidána byla kontextová nabídka, byla umožněna práce s uzly pomocí systémové schránky a klávesových zkratek. Strom rovněž zabraňuje přesunutí uzlu do nekompatibilní sekce a hlídá konzistenci projektu.

Pevnou strukturu stromu tvoří kategorie:

- Barvy
- Vektory
- Textury
- Modely
- Shadery
- Programy

Každá kategorie pak může obsahovat položky kompatibilního typu.

5.4 Komponenta pro editaci shaderů

Přestože XAML nabízí komponenty umožňující různé způsoby editace textu, v základu žádná z nich nezvládá pro editor potřebné zvýrazňování syntaxe. Při hledání vhodných kandidátů na komponentu editoru zdrojových kódů byla nalezena komponenta AvalonEdit.

Tato komponenta byla původně vytvořena pro potřeby vývojového prostředí SharpDevelop, sloužící k vývoji aplikací v jazyce C#. Komponenta AvalonEdit je dostupná pod svobodnou licencí **MIT**, umožňuje zvýrazňování syntaxe založené na datech nahrávaných ze souboru s definicí syntaxe, podporuje doplňování kódu a seskupování syntakticky souvisejících bloků textu. [43]

Použití komponenty je velmi jednoduché díky dobré dokumentaci a ukázkovým příkladům. S ohledem na původ je komponenta odladěná a velmi stabilní.

5.5 Modularita editoru

Zatímco strom projektu slouží k organizování a práci s uzly, prvky které uzly představující je možné upravovat skze editory panelu pracovní plochy. Ta je tvořena formou karet, kdy každá karta představuje editor jednoho uzlu stromu. Editaci uzlu je možné zahájit poklepáním na něj, stiskem klavesy enter, nebo případně skrze nabídky. Pro jeden uzel není možné otevřít více karet.

Aby bylo budoucí rozšiřování editoru o novou funkcionalitu co nejjednodušší, je celá část s editory řešena odděleně a modulárně. Základní stavební kámen tvoří třída EditorUserControl, která je sama potomkem třídy UserControl. Pro vytvoření nového typu editoru stačí rozšířit její dvě metody Load a Save.

Pro snažší práci se soubory a jejich linkování do projektu je k dispozici třída FileNameControl. Ta vedle klasického nalezení souboru skrze dialogová okna operačního systému umožňuje zadat soubor i přetažením myši, a to i na plochu editoru, na kterém se nachází.

6 Testování a zhodnocení

V rámci testování byl kladen důraz na to, aby všechny části editoru pracovaly konzistentně a nedocházelo k pádům či vyvolání neošetřených výjimek. Testována byla funkčnost i rozložení grafického rozhranní, a také výkon celé aplikace.

6.1 Testy uživatelského rozhranní

Vzniklý editor má sloužit jako výukový nástroj, a proto by měl být intuitivní a jednoduše pochopitelný. K ověření vhodného výběru rozložení vizuálních komponent byl použit nástroj pro zaznamenávání trajektorie pohybu myši, konkrétně **IOGraph** dostupný z webu *http://iographica.com/*. Vedle samotného pohybu byly zaznaménávány i informace o tom, jak dlouho strávil uživatel plněním dílčího úkolu.

Uživatel pracující s programem měl za úkol zpracovat následující úlohy, které představují základ práce s programem:

1. Nahrát do projektu model z externího souboru (byl dodán).
2. Vytvořit vlastní vertex a fragment shader (ukázka shaderů byla k dispozici).
3. Sestavit program stavající z modelu a shaderů.
4. Zobrazit výslednou scénu.

Testování provádělo několik dobrovolníků se znalostí informačních technologií na amatérské i profesionální úrovni. Všem byly dány základní instrukce, jak s programem pracovat. Pro ukázku byly vybrány dva vzorky, jejichž mapa pohybu myši je zobrazena na obrázku 9.

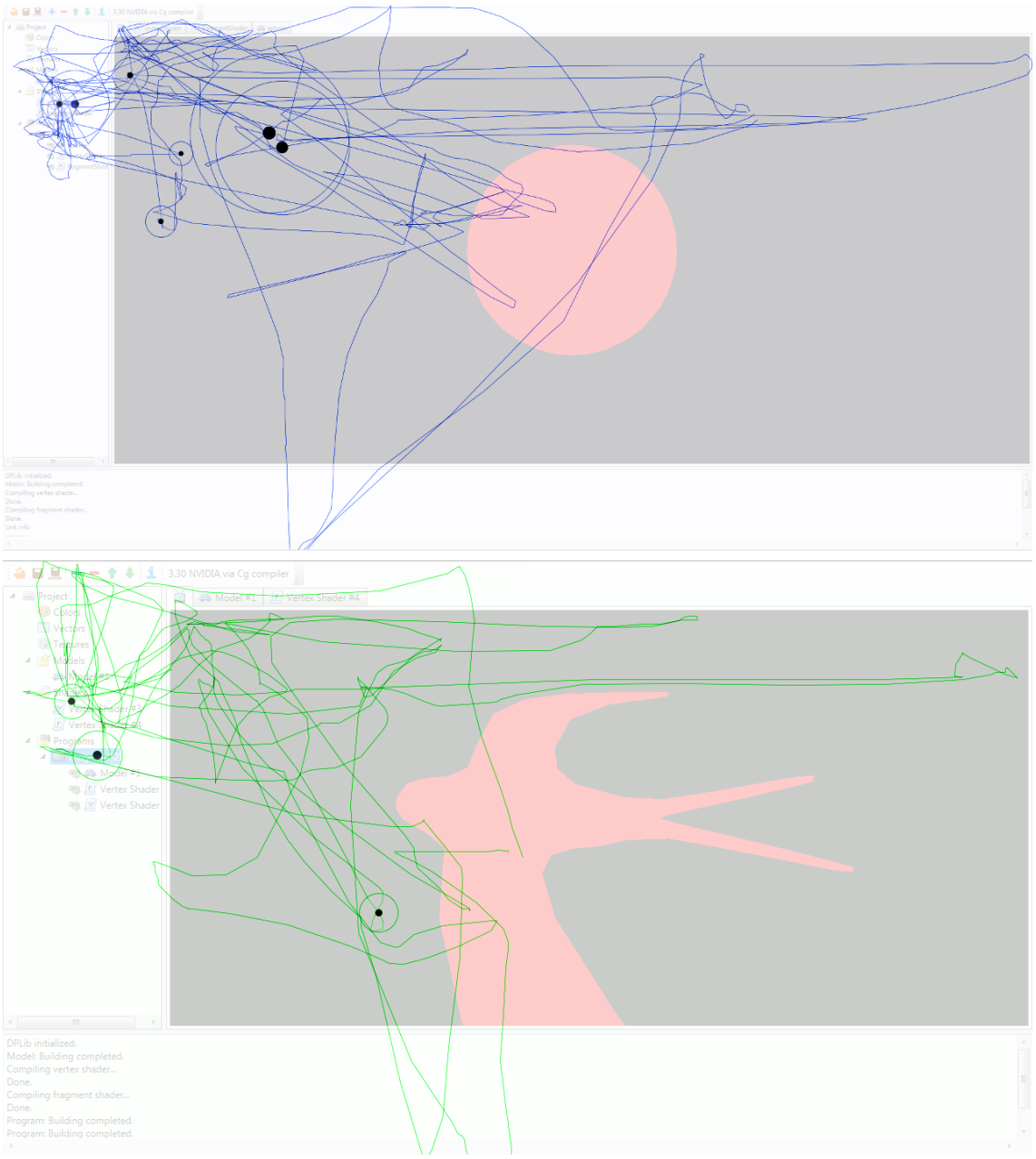
Na první obrazovce je zřetelně vidět, že nastavování všech prvků a práce se stromem projekt vyžaduje i při vysokém rozlišení monitoru minimální pohyb myši. Přestože testující uživatel nebyl v oboru počítačové grafiky odborně vzdělán, zabralo mu splnění všech úkolů pouhé 2 minuty.

Na druhé obrazovce je vidět téměř totožný průběh testování, přičemž uživatel se oborem počítačové grafiky zabývá na profesionální úrovni. I zde nezabralo testování více než několik málo minut.

Celkově dopadly testy uživatelského rozhranní dobře a ohlasy uživatelů na editor byly pozitivní. Díky implementaci klávesových zkratek, systémové schránky a přetahování prvků pomocí myši byla rychlost práce uživatelů s programem rychlejší, než se očekávalo.

6.2 Testy výkonu

V případě editoru shaderů nemá výkonnostní test vliv z pohledu samotného vykreslování obrazu grafickou kartou, jde pouze o měření rychlosti jednotlivých částí editoru a jejich schopnost reakce.



Obrázek 9: Pohyb myši během testování editoru

Rovněž se nepředpokládá, že by byl editor spouštěn na počítači, jehož výkon není úměrný současnému výkonnostnímu průměru. Jelikož je celá aplikace postavena na technologii .NET minimálně ve verzi 3.0, musí použitý počítač splňovat minimální požadavky tohoto frameworku. Pro praktický test spuštění editoru byl vybrán starší počítač (notebook), který by měl představovat pomyslnou hranici nutnou ke spuštění

aplikace. Konfigurace stroje byla následující:

- Procesor Intel® Core™ 2 Duo P7350 (2 GHz)
- Operační paměť 3 GB
- Grafická karta nVidia GeForce 9300M GS (256 MB)

Na výše zmíněném počítači byl opakovaně spuštěn editor, kompilovaný bez ladících informací, přičemž doba startu aplikace nepřesáhla 2 sekundy. Testy kompilace zadaných shaderů a nahrávání modelů či textur se pohybovaly v řádu stovek milisekund. Výkon editoru je tedy pro potřeby používání dostačující.

6.3 Shrnutí

Přestože editor je připraven pro praktické využití ve výuce nebo i mimo ni, možnosti jeho dalšího vývoje jsou otevřené, a do budoucna lze přidat mnoho dalších užitečných funkcí navíc. Vylepšen by mohl být parser formátu Wavefront OBJ, a to další možnosti tohoto formátu.

Finální verze, která je součástí této práce, má označení 0.9, což vyjadřuje připravenost k nasazení a pro povýšení na verzi 1.0 je potřeba, aby byl editor nějakou dobu využíván v reálném provozu.

7 Závěr

Před samotným započetím implementace editoru byly shromážděny různé technické údaje o všech potřebných technologiích, které budou v rámci editoru shaderů využity. Jazyk GLSL se ukázal jako nejlepší z možných řešení pro vývoj grafických shaderů, průzkum však ukázal, že pro něj neexistuje vhodný editor, vhodný i pro jeho výuku.

Výsledkem této práce je tedy plnohodnotný editor právě pro shadery jazyka GLSL. Program podporuje nejnovější specifikaci OpenGL i shaderovacího jazyka, a je navržen tak, aby do něj bylo možné jednoduše přidávat další funkce, které přinese budoucí vývoj normy OpenGL.

K realizaci samotného programu byl zvolen jazyk C#, pracující nad frameworkem .NET, v kombinaci se sadou komponent WPF a jazykem XAML. Neméně důležitou úlohu zde tvoří jazyk C, v němž je vytvořena podpůrná knihovna, která do celého řešení umožňuje připojit a pracovat s knihovnou OpenGL. V rámci práce byla vyvinuta nezávislá komponenta, umožňující pracovat s OpenGL jako s komponentou sady WPF. Volba technologie .NET přinesla editoru výhody ve formě lepší integrace se systémem Windows. Rozhraní editoru je tak pro uživatele přívětivé díky možnosti klávesových zkratek nebo přetahování objeků myší, což prokázalo i výsledné testování celé aplikace.

8 Reference

[1] *3D Graphics APIs* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.cg.tuwien.ac.at/~wimmer/apis/API_Summary.html

[2] *About the OpenGL Architecture Review Board Working Group* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.opengl.org/archives/about/arb/

[3] SHREINER, Dave. *OpenGL*: *průvodce programátora*. Vyd. 1. Brno: Computer Press, 2006, 679 s. ISBN 80-251-1275-6.

[4] *Selecting a Shading Language - OpenGL.org* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.opengl.org/wiki/Selecting_a_Shading_Language

[5] *Core Language (GLSL) - OpenGL.org* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://www.opengl.org/wiki/Core_Language_(GLSL)#Version

[6] *Programming Guide for HLSL (Windows)* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx

[7] *Detecting the Shader Model - OpenGL.org* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.opengl.org/wiki/Detecting_the_Shader_Model

[8] *ValveSoftware/ToGL · GitHub* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://github.com/ValveSoftware/ToGL

[9] *gtksourceview - Source code editing widge* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://git.gnome.org/browse/gtksourceview/tree/data/language-specs/glsl.lang

[10] *RenderMonkey™ Toolsuite | AMD* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/rendermonkey-toolsuite/

[11] *NVIDIA Nsight | NVIDIA* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.nvidia.com/object/nsight.html

[12] *NVIDIA Nsight Visual Studio Edition | NVIDIA Developer Zone* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://developer.nvidia.com/nvidia-nsight-visual-studio-edition

[13] *FX Composer | NVIDIA Developer Zone* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://developer.nvidia.com/fx-composer

[14] *GPU PerfStudio 2 | AMD* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://developer.amd.com/tools-and-sdks/graphics-development/gpu-perfstudio-2/

[15] John Kessenich, LunarG. *The OpenGL® Shading Language* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

[16] Randi J. Rost, Bill Licea-kane. *OpenGL Shading Language*. Vyd. 3. Addison-Wesley Professional, 2009, 792 s. ISBN 0-321-63763-1.

[17] KERNIGHAN, Brian, Dennis M. RITCHIE. *Programovací jazyk C*. Vyd. 1. Brno: Computer Press, 2008, 286 s. ISBN 80-251-0897-X.

[18] PRATA, Stephen. *Mistrovství v C++*. 3. aktualiz. vyd. Překlad Boris Sokol. Brno: Computer Press, 2007, 1119 s. ISBN 978-80-251-1749-1.

[19] HEROUT, Pavel. *Učebnice jazyka Java*. 5., rozš. vyd. České Budějovice: Kopp, 2010, 386 s. ISBN 978-80-7232-398-2.

[20] BUČEK, Petr. *Multimediální platforma pro Java SE a Android* [online]. 2012 [cit. 2014-05-01]. Dostupné z: http://dspace.vsb.cz/handle/10084/93323. Bakalářská práce. Vysoká škola báňská - Technická univerzita Ostrava. Vedoucí práce Ing. Mgr. Michal Krumnikl.

[21] *Overview – Python 3.4.0 documentation* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://docs.python.org/3/index.html

[22] *WPF - Mono* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.mono-project.com/WPF

[23] *.Net and C# release history - CodeProject* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.codeproject.com/Reference/696530/Net-and-Csharp-release-history

[24] *C# and OpenGL* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://social.msdn.microsoft.com/Forums/en-US/7903a8a0-40a3-4a6e-a79f-af16a173e129/c-and-opengl

[25] *GTK+ 3 Reference Manual: GTK+ 3 Reference Manual* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://developer.gnome.org/gtk3/stable/

[26] *Qt - About Us* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://qt.digia.com/about-us/

[27] *History - wxWidgets* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://www.wxwidgets.org/about/history/

[28] LOY, Marc a Robert ECKSTEIN. *Java Swing*. 2nd ed. Sebastopol, CA: O'Reilly, c2003, xxiv, 1252 p. ISBN 05-960-0408-7.

[29] *The SWT FAQ* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.eclipse.org/swt/faq.php

[30] *XAML ve WPF* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://msdn.microsoft.com/cs-cz/library/ms747122(v=vs.110).aspx

[31] *Seriál Grafické formáty - Root.cz* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.root.cz/serialy/graficke-formaty/

[32] *The Tao Framework | Free Development software downloads at SourceForge.net* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://sourceforge.net/projects/taoframework/

[33] *SharpGL - Home* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://sharpgl.codeplex.com/

[34] *Programming Guide for DDS (Windows)* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://msdn.microsoft.com/en-us/library/bb943991.aspx

[35] *Wavefront OBJ: Summary from the Encyclopedia of Graphics File Formats* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.fileformat.info/format/wavefrontobj/egff.htm

[36] *KhronosGroup/OpenCOLLADA · GitHub* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://github.com/khronosGroup/OpenCOLLADA/

[37] Mark Barnes and Ellen Levy Finch, Sony Computer Entertainment Inc.. *COLLADA – Digital Asset Schema Release 1.5.0* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.khronos.org/files/collada_spec_1_5.pdf

[38] *Norma IEEE 754 a příbuzní: formáty plovoucí řádové tečky - Root.cz* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky/#k02

[39] *JSON* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://json.org/json-cz.html

[40] *Unsafe Code and Pointers (C# Programming Guide)* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://msdn.microsoft.com/en-us/library/t2yzs44b.aspx

[41] *Walkthrough: Hosting a Windows Forms Control in WPF by Using XAML* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms742875(v=vs.110).aspx

[42] *"{Binding OpenGl To WPF}" : Part 2 « Planning the Spontaneous* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://slizerboy.wordpress.com/2010/02/03/binding-opengl-to-wpf-part-2/

[43] *AvalonEdit by icsharpcode* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://avalonedit.net/

[44] *RenderMonkey* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://developer.amd.com/wordpress/media/2012/10/FullScreen-RenderMonkey_lrg.jpeg

[45] *FX Composer* [online]. 2014 [cit. 2014-05-01]. Dostupné z: https://developer.nvidia.com/sites/default/files/akamai/gamedev/images/FXComposer_Flagship_Small.jpg

Přílohy

Přiložené CD obsahuje zdrojové kódy i spustitelný editor shaderů. Ke spuštění je potřeba operační systém Windows 7 nebo vyšší. Rovněž je potřeba mít nainstalován .NET framework alespoň ve verzi 3.0. Na CD je i elektronická verze této práce včetně zdrojových kódů v jazyce LaTeX.

Obsah CD

- **bin/** - spustitelný shader editor včetně ukázek
- **src/** - celé zdrojové kódy editoru
- **tex/** - zdrojové kódy této práce včetně obrázků
- **Diplomová práce.pdf** - samotná práce v elektronické podobě
- **Návod k použití.pdf** - stručná dokumentace k ovládání editoru